SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 2459

INTERAKTIVNI GRAFIČKI OBJEKTI NA WEB-U

Matejka Ivančić

Zagreb, lipanj 2012.

Sadržaj

1 Uvod

Od prve jednostavne web stranice s crnim tekstom koju je napravio Tim Berners Lee 1981., web grafika se naglo razvila do današnjeg obujma. Dobro poznata `<img>` oznaka je prvi put predložena 1993., i iako Web danas čini potpuno drugo okruženje, `<img>` oznaka još uvijek ostaje najčešći oblik prikaza slika [1].

Web je uvijek bio vizualni medij, ali vrlo ograničen. Do sada su web programeri bili ograničeni na CSS i JavaScript za izradu animacija i vizualnih efekata, ili su se morali osloniti na priključke (eng. *plug-in*) kao što je Flash. Nedavni dolazak novih tehnologija kao što su canvas element, Web GL i SVG slike otvaraju mogućnosti izrade naprednih grafičkih aplikacija bez potrebe dodatnih priključaka. Ove tri nove tehnologije sve više dobivaju podršku u modernim preglednicima (eng. *browser*), te se tako smanjuje potreba za priključcima [2].

Canvas element `<canvas>` je nova značajka HTML5 koja omogućuje crtanje grafike na jednostavan način koristeći JavaScript [3]. Canvas je bitmap sustav, što znači da se crta u obliku jedne 2D slike, te svaka promjena zahtjeva crtanje cijele nove slike. Koristi se za vizualizaciju podataka, animiranu grafiku, web aplikacije i igre.

Scalable Vector Graphics (SVG) je 2D vektorski zapis u XML formatu [4]. SVG specfikacija je otvoreni standard koji se razvija od 1999. Moderni preglednici pružaju osnovnu podršku za SVG, te se uz JavaScript biblioteke danas češće koriste.

WebGL (*Web Graphics Library*) je nova web tehnologija koja osigurava JavaScript API (eng. *Application Programming Interface*) za ostvarivanje prikaza 3D grafike unutar bilo kojeg kompatibilnog web preglednika bez uporabe priključaka [5]. WebGL proširuje mogućnosti HTML5 canvas elementa omogućujući crtanje u 3D .

U ovom radu ću proučiti tehnologije koje omogućuju prikaz i interaktivno rukovanje objektima na web-u. Posebno ću razmotriti canvas i KineticJS –

biblioteka koja omogućuje interaktivnost za canvas, te WebGL i jedan od njegovih biblioteka – Three.js.

2 Canvas

HTML5 specifikacija uključuje brojne nove značajke, od kojih je jedan `<canvas>` element. HTML5 `canvas` nudi jednostavan i moćan način crtanja koristeći JavaScript. Za razliku od nekih drugih tehnologija, za upotrebu canvas-a nisu potrebne dodatne biblioteke, već sve što je potrebno je HTML5 kompatibilan preglednik i *text editor* [6].

Canvas element se na jednostavan način može opisati kao područje na kojem se može crtati. To područje ima svoju visinu i duljinu (fiksna veličina), te koristi JavaScript kao medij koji crta i animira kompleksnu grafiku kao što su grafovi, kompozicije slika i izrada animacija [3].

Canvas element također koristi WebGL za sklopovski ubrzanu 3D grafiku web stranica. Više o WebGL-u u poglavlju 3.

2.1 Osnove korištenja

Kreiranje canvas konteksta jednako je jednostavno kao i dodavanje `<canvas>` elementa u HTML dokument:

```
<canvas id="canvas" width="300" height="300"></canvas>
```

Zadaje se ID elementa kako bi ga mogli pronaći u JavaScript kodu, te visina i duljina. Sada imamo prazan „papir“ za crtanje. Kako bi crtali unutar canvas elementa moramo koristiti JavaScript [3].

2.2 Ostvarivanje prikaza u kontekstu

`<canvas>` stvara fiksnu površinu za crtanje koja izlaže jedan ili više „kontekst“ za ostvarivanje prikaza (eng. *rendering contexts*) u koji se izdaju JavaScript naredbe za crtanje i manipuliranje prikazanog sadržaja. Preglednici mogu implementirati više canvas konteksta gdje različiti API-i osiguravaju funkcionalnost crtanja [7]. Unutar ovog poglavlja stavljen je fokus na 2D kontekst. Drugi konteksti mogu pružiti različite vrste ostvarivanje prikaza, kao npr. 3D kontekst ("*experimental-webgl*") temeljen na OpenGL ES.

`<canvas>` je inicijalno prazan. Kako bi prikazali nešto, skripta prvo mora pristupiti kontekstu i crtati na njemu. Canvas element ima DOM (eng. *Document Object Model*) metodu `getContext` koja se koristi za dobivanje konteksta i funkcija za crtanje. `getContext()` prima jedan parametar – tip konteksta.

```
1. var canvas = document.getElementById('canvas');
2. var ctx = canvas.getContext('2d');
```

U prvoj liniji se preuzima canvas čvor koristeći `getElementById` metodu. Nakon toga možemo pristupiti kontekstu za crtanje koristeći `getContext` metodu.

2.2.1 Jednostavan primjer

Jednostavan primjer crtanja s canvasom (prema [8]):

```
1. <html>
2.  <head>
3.   <script type="application/javascript">
4.       function draw() {
5.            var canvas = document.getElementById("canvas");
6.            var ctx = canvas.getContext("2d");
7.
8.            ctx.fillStyle = "rgb(200,0,0)";
9.            ctx.fillRect (10, 10, 55, 50);
10.
11.           ctx.fillStyle = "rgba(0, 0, 200, 0.5)";
12.           ctx.fillRect (30, 30, 55, 50);
13.      }
14.        </script>
15.  </head>
16.  <body onload="draw()">
17.     <canvas id="canvas" width="300" height="300"></canvas>
18.  </body>
19.  </html>
```

Funkcija `draw` uzima canvas element i dobiva `2d` kontekst. Objekt `ctx` tada može biti korišten za ostvarivanje prikaza canvas-a. Ovaj primjer jednostavno ispunjava dva kvadrata tako da postavlja `fillStyle` na dvije različite boje koristeći CSS specifikacije boje i pozivom `fillRect`. Drugi `fillStyle` koristi `rgba()` kako bi specificirao alfa vrijednost uz boju.

Rezultat:

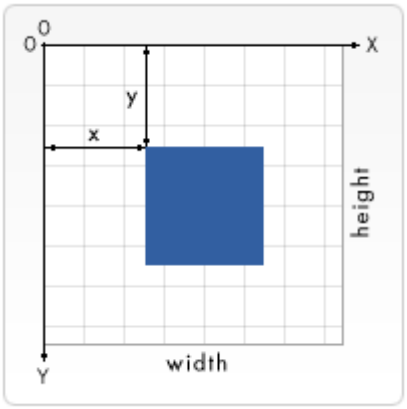
Slika 1. Primjer crtanja s canvas-om

Pozivi `fillRect`, `strokeRect` i `clearRect` crtaju ispunjeni, iscrtani ili čisti pravokutnik. Kako bi se prikazali kompleksniji oblici koriste se putovi (eng. *paths*).

2.3 Crtanje

2.3.1 Koordinatni sustav

Prije nego što se počne crtati, kratak pogled na canvas mrežu i raspon koordinata. Obično 1 jedinica u mreži odgovara 1 slikovnom elementu na canvasu. Ishodište sustava pozicionirano je u gornjem lijevom kutu, koordinata (0,0). Svi elementi su pozicionirani relativno s obzirom na ishodište.



Slika 2. Skica koordinatnog sustava

2.3.2 Crtanje oblika

Canvas podržava samo jedan primitivni oblik – pravokutnik. Svi ostali oblici se stvaraju kombiniranjem jedne ili više staza. Postoji kolekcija staza crtanje koje omogućuju kompoziciju vrlo kompleksnih oblika.

2.3.2.1 Pravokutnik

Postoje tri funkcije koje crtaju pravokutnik na canvasu:

`fillRect(x,y,width,height)` : crta ispunjeni pravokutnik
`strokeRect(x,y,width,height)` : crta obrub pravokutnika
`clearRect(x,y,width,height)` : obriše specificirano područje i čini ga transparentnim

```
1. function draw(){
2.   var canvas = document.getElementById('tutorial');
3.   if (canvas.getContext){
4.     var ctx = canvas.getContext('2d');
5.
6.     ctx.fillRect(25,25,100,100);
7.     ctx.clearRect(45,45,60,60);
8.     ctx.strokeRect(50,50,50,50);
9.   }
10.      }
```

Rezultat:

Slika 3. Primjer crtanja pravokutnika

`fillRect` funkcija crta najveći crni kvadrat dimezija 100×100 pixela. `clearRect` funkcija čisti kvadrat dimenzija 60×60 pixla od centra, te `strokeRect` crta obrub kvadrata 50×50 pixla unutar očišćeng kvadrata.

Za razliku od funkcija staza koje slijede, sve tri funkcije za crtanje pravokutnika se odmah crtaju na canvas.

2.3.3 Korištenje staza

Funkcija `beginPath` započinje novu stazu dok se `moveTo`, `lineTo`, `arcTo`, `arc` i slične metode koriste za dodavanje segmenata stazi. Segmenti se spajaju u listu pod-staza koje zajedno stvaraju oblik.

Kada je trenutna staza prazna i tek je pozvana funkcija `beginPath`, neovisno o kojoj je sljedećoj naredbi riječ, tretira se kao `moveTo()` – specificiranje početne pozicije.

Staza se zatvara koristeći `closePath`. Ova metoda zatvara oblik tako da crta ravnu liniju od trenutne pozicije do početne. Ako je oblik već zatvoren ili je samo jedna točka u listi, funkcija ne radi ništa.

Kada se oblik stvori, mogu se koristiti `fill` i/ili `stroke` za crtanje staze (oblik) na canvas. `stroke` se koristi za crtanje obruba, dok `fill` crta puni oblik. Kada se poziva `fill` metoda, staza će se automatski zatvoriti pa nije potrebno koristiti `closePath` metodu.

Primjer crtanja trokuta:

```
1. ctx.beginPath();
2. ctx.moveTo(75,50);
3. ctx.lineTo(100,75);
4. ctx.lineTo(100,25);
5. ctx.fill();
```

2.3.3.1 moveTo

Vrlo važna funkcija koja ne crta je `moveTo` funkcija. Ovu funkciju možemo zamisliti kao podizanje olovke s papira s jednog mjesta na drugo.

`moveTo(x, y)` – prima dva argumenta, x i y, koji su koordinate nove pozicije.

Kada se canvas inicijalizira ili se pozove `beginPath` metoda, potrebno je upotrijebiti `moveTo` metodu za pomicanje početne točke crtanja na novu poziciju. `moveTo` metoda se također koristi za crtanje nepovezanih staza.

`moveTo` *primjer*

```
1. ctx.beginPath();
2. ctx.arc(75,75,50,0,Math.PI*2,true); // Vanjski krug
3. ctx.moveTo(110,75);
4. ctx.arc(75,75,35,0,Math.PI,false);   // Usta (clockwise)
5. ctx.moveTo(65,65);
6. ctx.arc(60,65,5,0,Math.PI*2,true);  // Lijevo oko
7. ctx.moveTo(95,65);
8. ctx.arc(90,65,5,0,Math.PI*2,true);  // Desno oko
9. ctx.stroke();
```

Slika 4. Primjer korištenja `moveTo`

Desna slika pokazuje gdje se koristi `moveTo` metoda (crvene linije).

`arc` funkcija je obrađena kasnije.

2.3.3.2 Linije

Za crtanje ravne linije koristi se `lineTo` metoda.

`lineTo(x, y)` – metoda prima dva argumenta – x i y – koordinate krajnje točke linije. Početna točka je ovisna o prijašnjim nacrtanim stazama, pa je tako zadnja točka prijašnje staze početna točka sljedeće.

`lineTo` *primjer*

Primjer pokazuje dva nacrtana trokuta, jedan popunjen a drugi ocrtan. (Slika 5.)

Prvo se poziva metoda `beginPath` koja započinje novu stazu. Zatim se koristi moveTo metoda, kako bi se pomaknula početna točka.

Kao što je već spomenuto, kada se koristi `fill` metoda, staza se automatski zatvara, dok kod upotrebe `stroke` to nije tako. Da se nije upotrijebila `closePath()` metoda, nacrtale bi se samo dvije linije (stranice drugog trokuta), a ne potpuni trokut.

Slika 5. Primjer `lineTo` metode.

Kod primjera:

```
1. // Filled triangle
2.  ctx.beginPath();
3.  ctx.moveTo(25,25);
4.  ctx.lineTo(105,25);
5.  ctx.lineTo(25,105);
6.  ctx.fill();
7.
8. // Stroked triangle
9.  ctx.beginPath();
10. ctx.moveTo(125,125);
11. ctx.lineTo(125,45);
12. ctx.lineTo(45,125);
13. ctx.closePath();
14. ctx.stroke();
```

2.3.3.3 Lukovi

Za crtanje likova i krugova koristi se metoda arc. Specifikacija također opisuje arcTo metodu.

```
arc(x, y, radius, startAngle, endAngle, anticlockwise)
```

Metoda prima pet parametra: `x` i `y` – koordinate centra kružnice, radius, `startAngle` i `endAngle` definiraju početnu i završnu točku luka u radijanima i mjere se od x osi. `anticlockwise` parametar je tipa boolean. `True` crta luk obrnuto od kazaljke na satu, `false` crta u smjeru kazaljke na satu.

Kutovi u arc funkciji se mjere u radijanima, ne stupnjevima. Preračunavanje stupnjeva iz radijana može se dobiti pomoću sljedećeg JavaScript izraza:

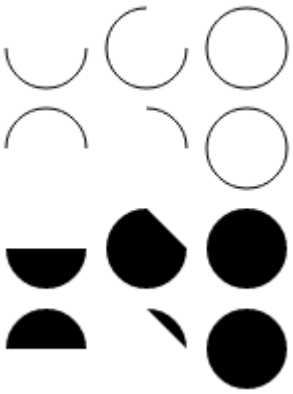
```
var radians = (Math.PI/180)*degrees.
```

`arc` *primjer*

Primjer pokazuje 12 različitih lukova, svi s različitih kutova i punjenja.

`for` petlje služe za prolaz kroz redove i stupce lukova. Svaki luk se stvara s novom stazom koristeći `beginPath`. Zatim su izlistani parametri. `radius` i `startAngle` su fiksni. `endAngle` počinje od 180° te se povečava za 90° dok ne formira puni krug (zadnji stupac). Prvi i treći red se crtaju u smjeru kazaljke na satu, dok drugi i četvrti ne. `if` izraz utječe na tip crtanja. Prva dva reda se crtaju sa `stroke`, a druga dva s `fill`.

```
1. for(var i=0;i<4;i++){
2.   for(var j=0;j<3;j++){
3.     ctx.beginPath();
4.     var x              = 25+j*50;               // x koordinata
5.     var y              = 25+i*50;               // y koordinata
6.     var radius         = 20;                    // Radijus luka
7.     var startAngle     = 0;                     // Početna točka
8.     var endAngle       = Math.PI+(Math.PI*j)/2; // Krajnja točka
9.     var anticlockwise  = i%2==0 ? false : true; // smjer crtanja
10.
11.      ctx.arc(x,y,radius,startAngle,endAngle, anticlockwise);
12.
13.      if (i>1){
14.        ctx.fill();
15.      } else {
16.        ctx.stroke();
17.      }
18.    }
19.  }
```



Slika 6. Primjer `arc`

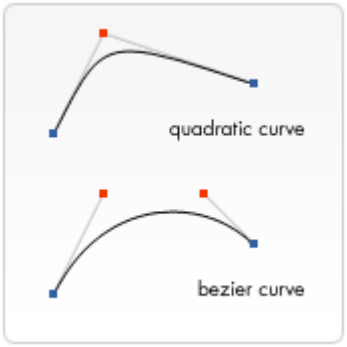
2.3.3.4 Bezierova i kvadratna krivulja

Za crtanje kompleksnih oblika najčešće se koristi Bezierova krivulja.

```
quadraticCurveTo(cp1x, cp1y, x, y)
```

```
bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)
```

Razlika ovih dviju krivulja se na jednostavan način može opisati pomoću sljedeće slike.



Slika 7. Bezierova i kvadratna krivulja

Kvadratna Bezierova krivulja ima početnu i završnu točku (plave točke) i samo jednu kontrolnu točku (crvena točka), dok kubna Bezierova krivulja koristi dvije kontrolne točke.

`x` i `y` parametri su u obje metode koordinate završne točke. `cp1x` i `cp1y` su koordinate prve kontrole točke, `cp2x` i `cp2y` su koordinate druge kontrolne točke.

Upotrebna Bezierovih krivulja nije najjednostavnija i može biti izazovna za korištenje jer za razliku od vektorskog crtanja u programima kao što su Adobe Illustrator, ne vidimo odmah rezultat, što čini crtanje kompleksnih oblika podosta teško.

`quadraticCurveTo` *primjer*

```
1. // Quadratric curves example
2.   ctx.beginPath();
3.   ctx.moveTo(75,25);
4.   ctx.quadraticCurveTo(25,25,25,62.5);
5.   ctx.quadraticCurveTo(25,100,50,100);
6.   ctx.quadraticCurveTo(50,120,30,125);
7.   ctx.quadraticCurveTo(60,120,65,100);
8.   ctx.quadraticCurveTo(125,100,125,62.5);
9.   ctx.quadraticCurveTo(125,25,75,25);
10.  ctx.stroke();
```

Slika 8. Primjer crtanja kvadratnom krivuljom

`bezierCurveTo` *primjer*

```
1. // Bezier curves example
2.       ctx.beginPath();
3.       ctx.moveTo(75,40);
4.       ctx.bezierCurveTo(75,37,70,25,50,25);
5.       ctx.bezierCurveTo(20,25,20,62.5,20,62.5);
6.       ctx.bezierCurveTo(20,80,40,102,75,120);
7.       ctx.bezierCurveTo(110,102,130,80,130,62.5);
8.       ctx.bezierCurveTo(130,62.5,130,25,100,25);
9.       ctx.bezierCurveTo(85,25,75,37,75,40);
10.      ctx.fill();
```

Slika 9. Primjer crtanja Bezierovom kubnom krivuljom

2.3.3.5 Pravokutnici

Uz već navedene metode crtanja pravokutnika, također postoji metoda `rect` koja dodaje pravokutnu stazu u listu staza.

```
rect(x, y, width, height)
```

Metoda prima 4 argumenta. `x` i `y` koordinate definiraju gornju lijevu točku pravokutnika. `width` i `height` definiraju duljinu i visinu pravokutnika.

Nakon što se metoda izvede, `moveTo` metoda se automatski pozove i postavi na (0,0) koordinatu (početna standardna vrijednost).

2.3.3.6 Stvaranje kombinacija

Do sada u primjerima je bilo pokazano korištenje samo jedne vrste funkcije po obliku, no ne postoji nikakva limitacija na broj tipova koji se može koristiti unutar staze kod stvaranja nekog oblika.

Slika 10. Primjer kombiniranja

```
1.  <html>
2.  <head>
3.   <script type="application/javascript">
4.       function draw() {
5.            var canvas = document.getElementById("canvas");
6.            var ctx = canvas.getContext("2d");
7.
8.            ctx.fillStyle = "red";
9.
10.           ctx.beginPath();
11.           ctx.moveTo(30, 30);
12.           ctx.lineTo(150, 150);
13.
14.           ctx.bezierCurveTo(60, 70, 60, 70, 70, 150);
15.           ctx.lineTo(30, 30);
16.           ctx.fill();
17.      }
18.  </script>
19.  </head>
20.  <body onload="draw()">
21.        <canvas id="canvas" width="300" height="300"></canvas>

22.  </body>
23.  </html>
```

2.3.4 Grafičko stanje

Atributi konteksta kao što su `fillStyle`, `strokeStyle`, `lineWidth` i `lineJoin` su dio trenutno grafičkog stanja (eng. `graphics state`). Canvas metode `save()` i `restore()` se koriste za spremanje i vraćanje canvas stanja. Canvas stanje je u osnovi snimka svih stilova i transformacija koje su primijenjene. Obje metode ne primaju parametre.

Canvas stanja se spremaju na stog. Svaki put kada se pozove `save` metoda, trenutno crtano stanje je gurnuto na vrh stoga. Crtano stanje se sastoji od:

- Primijenjenih transformacija (npr. translacija, rotacija, skaliranje)
- Vrijednosti `strokeStyle`, `fillStyle`, `globalAlpha`, `lineWidth`, `lineCap`, `lineJoin`, `miterLimit`, `shadowOffsetX`, `shadowOffsetY`, `shadowBlur`, `shadowColor`, `globalCompositeOperation` svojstva
- Trenutna staza rezanja (eng. *clipping path*)

Metoda `save` se može pozivati neograničeni broj puta. Svaki put kada se pozove `restore` metoda, zadnje spremljeno stanje se vraća sa stoga i sve spremljene postavke su vraćene.

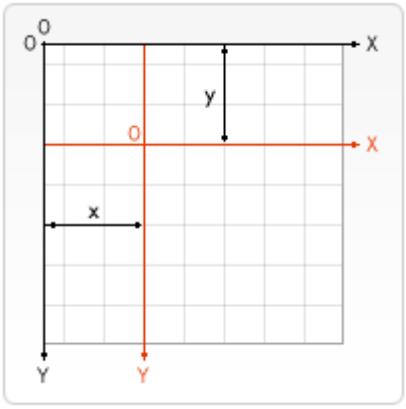
2.3.5 Transformacije

2.3.5.1 Translacija

Translacija se koristi za pomicanje ishodišta u koordinatnom sustavu.

```
translate(x, y)
```

Metoda prima dva argumenta. `x` je vrijednost pomaka u lijevo ili desno i `y` je vrijednost pomaka gore ili dolje.



Slika 11. Ilustracija translacije

Korisno je spremiti canvas stanje prije bilo koje transformacije jer je u većini slučajeva jednostavnije pozvati metodu `restore` od ručnog vraćanja stanja.

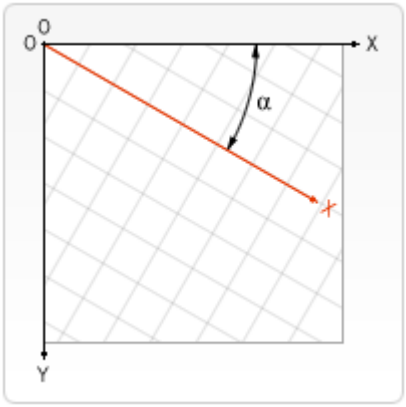
Korisnost translacije vidimo u primjerima gdje crtamo oko ishodišta, pa bi bez translacije vidjeli samo četvrtinu slike. Također metoda translacije daje slobodu crtanja bilo gdje na canvasu bez ručnog prilagođavanja koordinata.

2.3.5.2 Rotacija

Rotacija se koristi za rotiranje canvasa oko ishodišta.

```
rotate(angle)
```

Metoda prima samo jedan argument – kut rotacije. Rotacija je u smjeru kazaljke na satu izražena u radijanima.



Slika 12. Rotacija canvasa

Centar rotacije je uvijek ishodište canvasa. Kako bi promijenili centar rotacije, potrebno je translatirati canvas u željenu točku.

2.3.5.3 Skaliranje

Skaliranje se koristi za povećanje ili smanjenje jedinica u canvas sustavu te se tako mogu crtati uvećani ili smanjeni oblici.

```
scale(x, y)
```

Metoda prima dva parametra, `x` je faktor skaliranja u horizontalnom smjeru i `y` koji je faktor skaliranja u vertikalnom smjeru. Oba parametra moraju biti realni brojevi, ne nužno pozitivni. Vrijednosti manje od 1.0 smanjuju veličinu jedinice, vrijednosti veće od 1.0 uvećaju veličinu jedinice. Vrijednost 1.0 nema utjecaja na skaliranje. Upotreba negativnih brojeva stvara zrcaljenje prema osi.

Po definiciji jedna jedinica na canvas-u je točno jedan slikovni element. Ako se npr. primijeni skaliranje s faktorom 0.5, rezultirana jedinica bi postala 0.5 pixela, te bi oblici bili crtani u pola veličine. Jednako tako skaliranje s faktorom 2.0 bi

povećalo jedinicu veličine u dva slikovna elementa, te bi crtani objekti bili dvostruko veći.

2.3.5.4 transform metoda

Zadnja metoda transformiranja dopušta direktno modificiranje transformacijske matrice.

`transform(m11, m12, m21, m22, dx, dy)`

Metoda množi trenutnu matricu transformacije s opisanom matricom:

$$\begin{bmatrix} m11 & m21 \\ m12 & m22 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Ako je ikoji od argumenata beskonačan, transformacijska matrica se označava kao beskonačnost, bez da baca iznimku.

`setTransform(m11, m12, m21, m22, dx, dy)`

Ova metoda resetira trenutnu transformacijsku matricu u matricu identiteta, a zatim poziva transform metodu s istim zadanim argumentima.

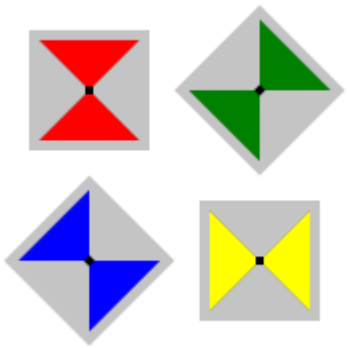
Primjer

Sljedeći primjer koristi staze, grafička stanja i trenutnu transformacijsku matricu. Metode konteksta `translate()`, `scale()` i `rotate()` transformiraju trenutnu matricu. Sve iscrtane točke su prvo transformirane tom matricom.

```
1. function drawBowtie(ctx, fillStyle) {
2.   ctx.fillStyle = "rgba(200,200,200,0.3)";
3.   ctx.fillRect(-30, -30, 60, 60);
4.
5.   ctx.fillStyle = fillStyle;
6.   ctx.globalAlpha = 1.0;
7.   ctx.beginPath();
8.   ctx.moveTo(25, 25);
9.   ctx.lineTo(-25, -25);
10.  ctx.lineTo(25, -25);
11.  ctx.lineTo(-25, 25);
12.  ctx.closePath();
13.  ctx.fill();
14. }
15.
16. function dot(ctx) {
17.   ctx.save();
18.   ctx.fillStyle = "black";
19.   ctx.fillRect(-2, -2, 4, 4);
20.   ctx.restore();
21. }
```

```
22.
23. function draw() {
24.   var canvas = document.getElementById("canvas");
25.   var ctx = canvas.getContext("2d");
26.
27.   ctx.translate(45, 45);
28.
29.   ctx.save();
30.   drawBowtie(ctx, "red");
31.   dot(ctx);
32.   ctx.restore();
33.
34.   ctx.save();
35.   ctx.translate(85, 0);
36.   ctx.rotate(45 * Math.PI / 180);
37.   drawBowtie(ctx, "green");
38.   dot(ctx);
39.   ctx.restore();
40.
41.   ctx.save();
42.   ctx.translate(0, 85);
43.   ctx.rotate(135 * Math.PI / 180);
44.   drawBowtie(ctx, "blue");
45.   dot(ctx);
46.   ctx.restore();
47.
48.   ctx.save();
49.   ctx.translate(85, 85);
50.   ctx.rotate(90 * Math.PI / 180);
51.   drawBowtie(ctx, "yellow");
52.   dot(ctx);
53.   ctx.restore();
54. }
```

Rezultat:



Slika 13. Primjer transformacija

Definirane su dvije metode `drawBowtie` i `dot` koje se pozivaju 4 puta. Prije svakog poziva, `translate()` i `rotate()` se koriste za postavljanje trenutne transformacijske matrice, koja pozicionira `dot` i `bowtie`. `Dot` iscrtava mali kvadrat centriran u `(0,0)`. Točka se pomiče pomoću transformacijske matrice.

`drawBowtie` iscrtava jednostavni put mašne (*bowtie path*) pomoću priloženog stila za ispunjavanje.

Kako su operacije nad matricama kumulativne, `save()` i `restore()` se koriste kod svakog skupa poziva kako bi se vratilo originalno canvas stanje. Važno je skrenuti pažnju kod rotacije koja se uvijek radi oko trenutnog centra, pa tako slijed `translate() rotate() translate()` će dati drugačiji rezultat od `translate() translate() rotate()`.

2.4 KineticJS

KineticJS je HTML5 Canvas JavaScript biblioteka koja proširuje 2D kontekst omogućujući canvas interaktivnost za desktop i mobilne aplikacije [9].

Crtanje vlastitih oblika ili slika pomoću postojećeg canvas API-a proširuje se dodavanjem osluškivača događaja (eng. *event listener*), pomicanjem, skaliranjem i rotacijom objekata nezavisno od drugih oblika i podrška animacija visokih performansi.

2.4.1 Kako KineticJS radi?

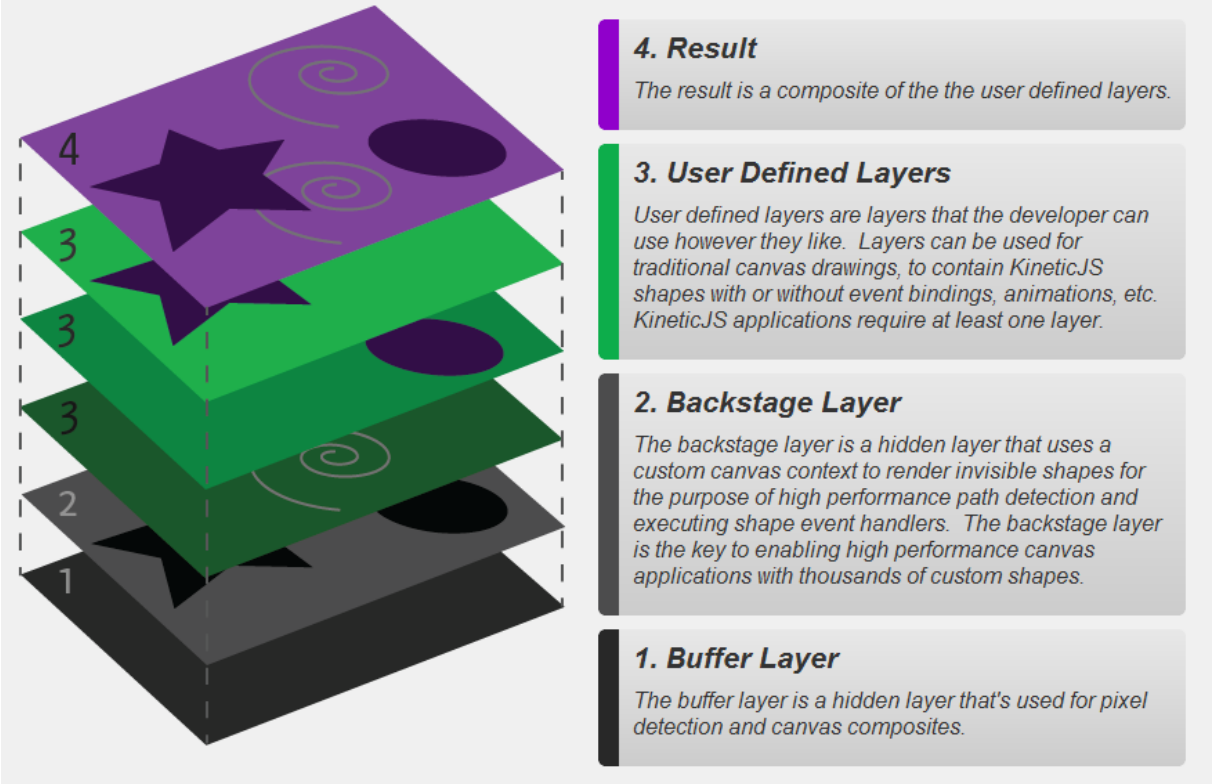
KineticJS aplikacije zahtijevaju da kontejner DOM element u HTML stranici sadrži pozornicu sastavljenu od slojeva (eng. *layers*). Svaki sloj je povezan sa svojim vlastitim canvas elementom i može sadržavati oblike. Oblicima, grupama i pozornicama se mogu dodavati osluškivači događaja, mogu se pomicati, rotirati, skalirati, slagati u slojeve... Grupe se mogu koristiti za sadržavanje oblika ili drugih grupa.

2.4.2 Kako KineticJS dobiva svoju brzinu?

Za rukovanje događajima, KineticJS pozornica je sastavljena od stražnjeg sloja i sloja spremnika (eng. *buffer layer*) za osiguravanje visoke performanse staza/putova i *pixel* detekcije. Animacije, tranzicije, te primi i pusti (eng. *drag and drop*) operacije su posebno glatke jer programeri/developeri mogu kreirati neograničeni broj korisničkih slojeva koji dozvoljavaju ažuriranje i ponovno crtanje nekih oblika bez diranja drugih. Dodatno, KineticJS prilagođava brzinu crtanja slika ovisno o CPU potražnji (eng. *frame rate*), koristi raspršivanje slojeva (eng. *layer*

hashing) kako bi se eliminiralo nepotrebno ponovno crtanje slojeva i prigušivanje slojeva (eng. *layer throttling*) za sprečavanje viška ponovnog crtanja. [9]

1. Sloj spremnika je skriveni sloj koji se koristi za detekciju slikovnih elemenata i canvas kompozicije.
2. Sloj iza pozornice je skriveni sloj koji koristi prilagođeni canvas kontekst za crtanje nevidljivih oblika radi visokih performansi detekcija staza i izvršavanje rukovatelja događaja oblika. Sloj iza pozornice je ključan za omogućavanje visoke performanse canvas aplikacija s tisuću vlastitih oblika.
3. Korisnički slojevi su slojevi koje programeri mogu koristiti kako god hoće. Slojevi mogu biti korišteni za tradicionalne canvas crteže, KineticJS oblike sa ili bez veza na događaje, animacije itd. KineticJS aplikacije zahtijevaju barem jedan sloj.
4. Rezultat je mješavina korisničkih slojeva.



Slika 14. Shema KineticJS

2.4.3 Crtanje oblika

Crtanje oblika, bilo to pravokutnik, krug, slika, tekst, linija, poligon ili neki prilagođeni kompleksniji oblik... zadaju se na sličan način. Kako bi se stvorio neki oblik, mora se instancirati njegov objekt sa željenim svojstvima (konfiguracija objekta) [10].

Primjer crtanja pravokutnika

```
1. <!DOCTYPE HTML>
2. <html>
3.   <head>
4.    <script src="kinetic-v3.9.3.js"></script>
5.    <script>
6.       window.onload = function() {
7.           var stage = new Kinetic.Stage({
8.           container: "container",
9.           width: 578,
10.          height: 200
11.       });
12.
13.       var layer = new Kinetic.Layer();
14.
15.       var rect = new Kinetic.Rect({
16.           x: 239,
17.           y: 75,
18.           width: 100,
19.           height: 50,
20.           fill: "#00D2FF",
21.           stroke: "black",
22.           strokeWidth: 4
23.        });
24.
25.        // add the shape to the layer
26.        layer.add(rect);
27.
28.        // add the layer to the stage
29.        stage.add(layer);
30.       };
31.   </script>
32.  </head>
33.  <body>
34.     <div id="container"></div>
35.  </body>
36. </html>
```

Slika 15. Nacrtani pravokutnik

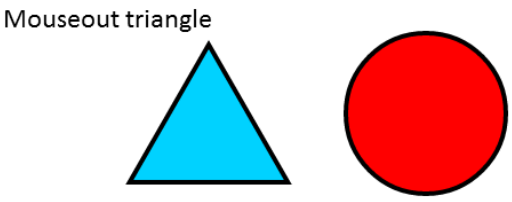
2.4.4 Događaji

2.4.4.1 Detekcija puta/staze

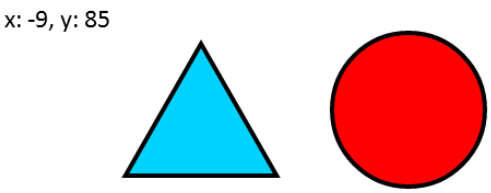
Kako bi detektirali događaje s KineticJS, koristi se `on()` metoda koja veže rukovatelj događaja s objektom oblika. `on()` metoda zahtijeva tip događaja i funkciju koja će se izvoditi kada dođe do događaja. KineticJS podržava `mouseover`, `mouseout`, `mousemove`, `mousedown`, `mouseup`, `click`, `dblclick`, `dragstart`, i `dragend` desktop događaje [11]. Po definiciji oblici su detektirani pomoću detekcije staze.

```
1. <script>
2.   shape.on('click', function(evt){
3.      // do stuff
4.   });
5. </script>
```

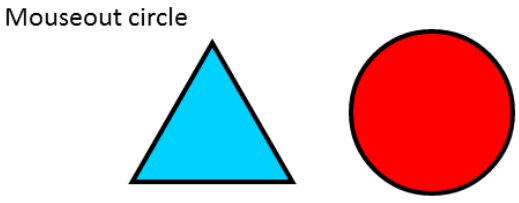
Primjer



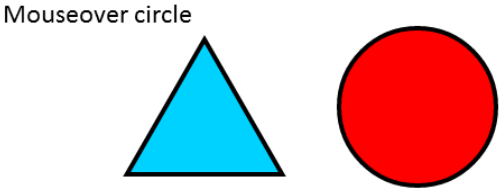
Slika 16. Primjer: miš izvan trokuta



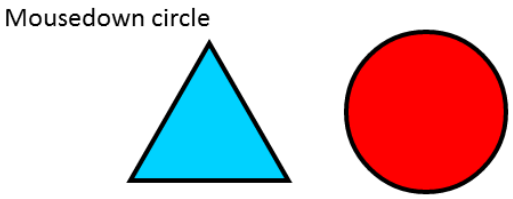
Slika 17. Primjer: miš unutar trokuta



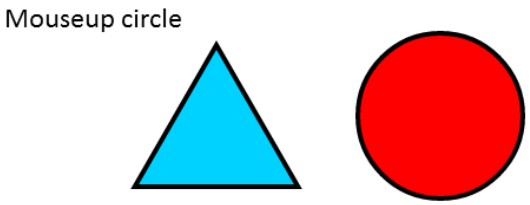
Slika 18. Primjer: miš izvan kruga



Slika 19. Primjer: miš unutar trokuta



Slika 20. Primjer: miš pritisnut na krug



Slika 21. Primjer: miš otpušten s kruga

Kod:

```
1. function writeMessage(messageLayer, message) {
2.         var context = messageLayer.getContext();
3.         messageLayer.clear();
4.         context.font = "18pt Calibri";
5.         context.fillStyle = "black";
6.         context.fillText(message, 10, 25);
7.       }
8.
9. window.onload = function() {
10.      var stage = new Kinetic.Stage({
11.                container: "container",
12.                width: 578,
13.                height: 200
14. });
15.
16. var shapesLayer = new Kinetic.Layer();
17. var messageLayer = new Kinetic.Layer();
18.
19. var triangle = new Kinetic.RegularPolygon({
20.                x: 190,
21.                y: 120,
22.                sides: 3,
23.                radius: 80,
24.                fill: "#00D2FF",
25.                stroke: "black",
26.                strokeWidth: 4
27.              });
28.
29. triangle.on("mouseout", function() {
30.                writeMessage(messageLayer, "Mouseout triangle");
31. });
32.
33. triangle.on("mousemove", function() {
34.             var mousePos = stage.getMousePosition();
35.             var x = mousePos.x - 190;
36.             var y = mousePos.y - 40;
37.           writeMessage(messageLayer, "x: " + x + ", y: " + y);
38.  });
39.
40.  shapesLayer.add(triangle);
41.
42.  var circle = new Kinetic.Circle({
43.                x: 380,
44.                y: stage.getHeight() / 2,
45.                radius: 70,
```

```
                fill: "red",
                stroke: "black",
                strokeWidth: 4
    });

    circle.on("mouseover", function() {
        writeMessage(messageLayer, "Mouseover circle");
    });
    circle.on("mouseout", function() {
        writeMessage(messageLayer, "Mouseout circle");
    });
    circle.on("mousedown", function() {
        writeMessage(messageLayer, "Mousedown circle");
    });
    circle.on("mouseup", function() {
        writeMessage(messageLayer, "Mouseup circle");
    });

    shapesLayer.add(circle);

    stage.add(shapesLayer);
    stage.add(messageLayer);
};
```

Moguće je vezati više događaja s jednim rukovateljem tako da se izlistaju događaji s jednim razmakom između [12].

```
<script>
  shape.on('mousedown mouseover' function(evt) {
    // do something
  });
</script>
```

2.4.4.2 Detekcija slikovnih elemenata

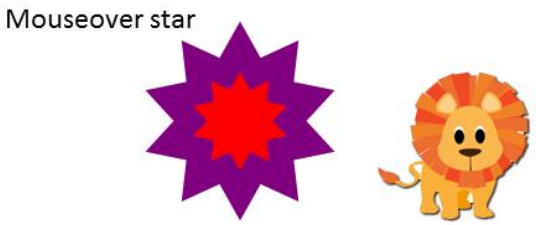
Kada želimo detektirati oblike i slike koristeći detekciju slikovnih elemenata sa KineticJS, mora se postaviti `detectionType` svojstvo konfiguracije objekta u `pixel` [13].

Kada se koristi pixel detekcija, potrebno je ručno spremiti podatke oblika koristeći `saveData()` metodu. Isto tako se mogu očistiti podaci oblika uz `clearData()` metodu.

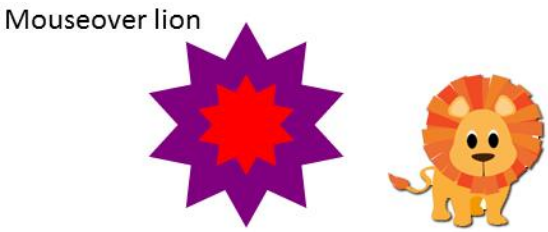
```
<script>
  var image = new Kinetic.Image({
    detectionType: 'pixel'
  });

  // save image data
  image.saveData();

```

```
9.   // clear image data
10.  image.clearData();
11. </script>
```

Primjer:



Slika 22. Primjer: miš preko zvijezde



Slika 23. Primjer: miš preko lava

```
1.       function writeMessage(messageLayer, message) {
2.         var context = messageLayer.getContext();
3.         messageLayer.clear();
4.         context.font = "18pt Calibri";
5.         context.fillStyle = "black";
6.         context.fillText(message, 10, 25);
7.       }
8.       function loadImage(stage, shapesLayer, messageLayer) {
9.         var imageObj = new Image();
10.              imageObj.onload = function() {
11.                var lion = new Kinetic.Image({
12.                  x: 300,
13.                  y: 50,
14.                  image: imageObj,
15.                  detectionType: "pixel"
16.                });
17.
18.                lion.on("mouseover", function() {
19.                  writeMessage(messageLayer, "Mouseover lion");
20.                });
21.
22.                lion.on("mouseout", function() {
23.                  writeMessage(messageLayer, "");
24.                });
25.
26.                shapesLayer.add(lion);
27.
28.                shapesLayer.draw();
29.
30.                lion.saveData();
31.      };
32.  imageObj.src = "lion.png";
33.  }
34.
35.            window.onload = function() {
36.              var stage = new Kinetic.Stage({
37.                container: "container",
38.                width: 578,
39.                height: 200
40.              });
41.              var shapesLayer = new Kinetic.Layer();
```

```
42.             var messageLayer = new Kinetic.Layer();
43.
44.             var star = new Kinetic.Star({
45.               x: 200,
46.               y: stage.getHeight() / 2,
47.               points: 10,
48.               innerRadius: 40,
49.               outerRadius: 60,
50.               fill: "red",
51.               stroke: "purple",
52.               strokeWidth: 20,
53.               detectionType: "pixel"
54.             });
55.
56.             star.on("mouseover", function() {
57.               writeMessage(messageLayer, "Mouseover star");
58.             });
59.
60.             star.on("mouseout", function() {
61.               writeMessage(messageLayer, "");
62.             });
63.
64.             shapesLayer.add(star);
65.             stage.add(shapesLayer);
66.             stage.add(messageLayer);
67.
68.             //save star pixel data so that it can be detected
69.             star.saveData();
70.
71.             loadImage(stage, shapesLayer, messageLayer);
72.           };
```

2.4.4.3 Povuci i pusti operacija

Za omogućavanje povuci i pusti operacije sa KineticJS potrebno je svojstvo `draggable` postaviti u `true` (istinito) kod inicijalizacije objekta oblika ili možemo koristiti `draggable()` metodu. `draggable()` metoda omogućuje povuci i pusti operacije i za desktop i za mobilne aplikacije automatski [14].

```
1. <script>
2.   // make draggable on instantiation
3.   var shape = new Kinetic.Circle({
4.     draggable: true;
5.   });
6.
7.   // make draggable after instantiation
8.   shape.draggable(true);
9. </script>
```

Primjer

```
1.          var stage = new Kinetic.Stage({
2.            container: "container",
3.            width: 578,
4.            height: 200
5.          });
6.          var layer = new Kinetic.Layer();
7.          var rectX = stage.getWidth() / 2 - 50;
8.          var rectY = stage.getHeight() / 2 - 25;
9.
10.               var box = new Kinetic.Rect({
11.                 x: rectX,
12.                 y: rectY,
13.                 width: 100,
14.                 height: 50,
15.                 fill: "#00D2FF",
16.                 stroke: "black",
17.                 strokeWidth: 4,
18.                 draggable: true
19.               });
20.
21.               // add cursor styling
22.               box.on("mouseover", function() {
23.                 document.body.style.cursor = "pointer";
24.               });
25.               box.on("mouseout", function() {
26.                 document.body.style.cursor = "default";
27.               });
28.
29.               layer.add(box);
30.               stage.add(layer);
```

2.4.5 Slojevi

Mijenjanje poretka korisničkih slojeva sa KineticJS može se pomoću sljedećih metoda: `moveToTop()`, `moveToBottom()`, `moveUp()`, `moveDown()`, ili `setZIndex()`, [15].

```
1.   shape.moveToTop();
2.   shape.moveToBottom();
3.   shape.moveUp();
4.   shape.moveDown();
5.   shape.setZIndex(5);
```

2.4.6 Prijelazi

2.4.6.1 Linearan prijelaz

Kreiranje linearnih prijelaza s KineticJS je jednostavno uz upotrebu metoda `transitionTo()`. Potrebno je odrediti svojstva prijelaza. Po definiciji, tip prijelaza je linearan. Svako numeričko svojstvo oblika, grupe, sloja ili pozornice može imati prijelaz kao što su `x,y,rotacija,visina, duljina, radijus, debljina crte, alfa vrijednost, veličina` itd. Konfiguracija postavljena u `transitionTo()` metodi mora imati određenu duljinu u sek koja definira koliko dugo prijelaz mora trajati [16].

```
1. shape.transitionTo({
2.   x: 100,
3.   duration: 1
4. });
```

Primjer

```
1. var stage = new Kinetic.Stage({
2.           container: "container",
3.           width: 578,
4.           height: 200
5.         });
6.         var layer = new Kinetic.Layer();
7.         var rect = new Kinetic.Rect({
8.           x: 100,
9.           y: 100,
10.               width: 100,
11.               height: 50,
12.               fill: 'green',
13.               stroke: 'black',
14.               strokeWidth: 2,
15.               alpha: 0.2
16.             });
17.
18.             layer.add(rect);
19.             stage.add(layer);
20.
21.             /*
22.              * wait one second before starting the
23.              * transition
24.              */
25.             setTimeout(function() {
26.               rect.transitionTo({
27.                 x: 400,
28.                 y: 30,
29.                 rotation: Math.PI * 2,
30.                 alpha: 1,
31.                 strokeWidth: 6,
32.                 scale: {
```

```
                    x: 1.3,
                    y: 1.3
                  },
                  duration: 1,
                });
              }, 1000);
```

2.4.6.2 Ostali prijelazi

Druge vrste prijelaza spadaju pod grupu prijelaza koja se zove *easing* prijelazi [17]. S KineticJS se ove vrste prijelaza koriste preko easing svojstva čvora prijelaza. *Easing* prijelazi su: `linear`, `ease-in`, `ease-out`, `ease-in-out`, `back-ease-in`, `back-ease-out`, `back-ease-in-out`, `elastic-ease-in`, `elastic-ease-out`, `elastic-ease-in-out`, `bounce-ease-in`, `bounce-ease-out`, `bounce-ease-in-out`, `strong-ease-in`, `strong-ease-out`, i `strong-ease-in-out`.

```
shape.transitionTo({
    x: 100,
    duration: 1,
    easing: 'bounce-ease-out'
});
```

2.4.6.3 Zaustavljanje i nastavljanje prijelaza

Svaki prijelaz se može zaustaviti i nastaviti pomoću KineticJS i njegovih `stop()` i `resume()` metoda objekta.

```
var trans = shape.transitionTo({
    x: 100,
    duration: 1,
    easing: 'bounce-ease-out'
});

  // stop transition
  trans.stop();

  // resume transition
  trans.resume();
```

3 WebGL

WebGL (Web Graphics Library) je JavaScript API temeljen na OpenGL ES 2.0 [5]. Omogućuje prikaz nevjerojatne 3D grafike unutar bilo kojeg kompatibilnog web preglednika bez korištenja dodataka (eng. *plug-in*). WebGL programi se sastoje od kontrolnog koda napisanog u JavaScriptu i koda za sjenčanje (eng. *shader*) koji se izvršava na grafičkoj jedinici računala (GPU, Graphics Processing Unit). WebGL koristi HTML5 Canvas element i pristupa mu se koristeći DOM sučelje.

Zato što se WebGL temelji na OpenGL-u i biti će integriran u popularnim preglednicima, nudi brojne prednosti kao što su (prema [18]):

- API temeljen na poznatom i široko prihvaćenom 3D grafičkom standardu
- Kompatibilnost preko više preglednika i platformi
- Uska integracija s HTML sadržajem uključujući slojevita kompozicija, interakcija s drugim HTML elementima i korištenje standardnih HTML mehanizama za rukovanje događajima
- Sklopovski ubrzana 3D grafika za okolinu preglednika
- Skriptna okolina koja olakšava prototipiranje 3D grafike – nije potrebno prevođenje i povezivanje prije gledanja i ispravljanje grešaka prikaza grafike

No WebGL je API niske razine (eng. *low-level*), pa tako i jednostavne stvari u WebGL-u rezultiraju u dobroj količini koda. Potrebno je učitati, kompajlirati i povezati programe za sjenčanje (eng. *shader*), postaviti varijable koje se prosljeđuju programima za sjenčanje, i izvoditi matematiku matrica za animaciju oblika. Tako je potreban preduvjet poznavanje

- GLSL, jezik za sjenčanje korišten u OpenGL-u i WebGL-u
- Računanje matrica za transformacije
- Grafički spremnici (eng. *vertex buffer*) koji sadrže informacije o pozicijama vrhova, normala, boja i tekstura

U poglavlju 3.6 pokazano je korištenje Three.js biblioteke za WebGL koja uvelike olakšava crtanje u WebGL-u.

3.1 Priprema za prikaz u 3D-u

Prva stvar koja je potrebna za korištenje WebGL-a za 3D prikaz je `canvas`. Sljedeći kod uspostavlja canvas i postavlja rukovatelja `onload` događaja koji će se koristiti za inicijalizaciju WebGL konteksta [19].

```
1. <body onload="start()">
2.   <canvas id="glcanvas" width="640" height="480">
3.     Vaš preglednik ne podržava HTML5
   <code>&lt;canvas&gt;</code>.
4.   </canvas>
5. </body>
```

3.1.1 Priprema WebGL konteksta

Funkcija `start()` je naš JavaScript kod koji se poziva nakon što se dokument učita. Njezin zadatak je postavljanje WebGL konteksta i iscrtavanje sadržaja.

```
1. function start() {
2.   var canvas = document.getElementById("glcanvas");
3.
4.   initWebGL(canvas);       // inicijalizacija GL konteksta
5.
6.   // Nastaviti samo ako je WebGL dostupan i radi
7.
8.   if (gl) {
9.     gl.clearColor(0.0, 0.0, 0.0, 1.0); // Postavi boju za
   čišćenje
10.    gl.enable(gl.DEPTH_TEST);        // Omogućiti testianje
   dubine
11.    gl.depthFunc(gl.LEQUAL);    // Bliže stvari skrivaju dalje
12.   gl.clear(gl.COLOR_BUFFER_BIT|gl.DEPTH_BUFFER_BIT);
13.  }
14. }
```

Prva stvar koju radimo je da dohvaćamo referencu na canvas i stavljamo je u varijablu `canvas`. Kada nam nije potrebno pristupanje canvas referenci u više navrata, poželjno je spremanje u lokalnu varijablu radije od globalne.

Nakon što imamo canvas, poziva se funckija initWebGL() koja inicijalizira WebGL kontekst (kod slijedi).

Ako je kontekst uspješno inicijaliziran, `gl` je referenca na njega. U ovom primjeru postavlja se boja za brisanje u crnu, nakon čega čistimo kontekst. Postavljamo

parametre konfiguracije konteksta: omogućujemo testiranje dubine i specificiramo da bliži objekti skrivaju/zasjenjuju objekte koji su dalje.

```
1. function initWebGL(canvas) {
2.   // Iniciijalizacija globalne varijable gl
3.   gl = null;
4.
5.   try {
6.     gl = canvas.getContext("webgl") ||
7.            canvas.getContext("experimental-webgl");
8.   }
9.   catch(e) {}
10.  if (!gl) {
11.       alert("Unable to initialize WebGL. Your browser may not s
   upport it.");
12.  }
13. }
```

Ime konteksta „experimental-webgl“ je privremeno ime za kontekst za vrijeme razvoja specifikacije. Ime „webgl“ će se koristiti nakon što su specifikacije finalizirane.

Nakon što se uspješno kreira WebGL kontekst može se početi crtati u njega.

3.2 Osvjetljavanje scene

Kako sada crtamo u 3D prostoru, potrebno je postaviti programe za sjenčanje koji osvjetljavaju scenu i crtaju objekt [20].

3.2.1 Inicijalizacija programa za sjenčanje

Programi za sjenčanje su programi koji govore WebGL-u gdje treba crtati i što. Programi za sjenčanje su specificirani s OpenGL ES Shading Language (GLSL). Kako bi jednostavnije održavali i ažurirali svoj sadržaj, možemo napisati kod koji učita programe za sjenčanje tako da ih pronađe u HTML dokumentu, umjesto građenja svega u JavaScript-u.

```
1. function initShaders() {
2.   var fragmentShader = getShader(gl, "shader-fs");
3.   var vertexShader = getShader(gl, "shader-vs");
4.
5.   // Create the shader program
6.
7.   var shaderProgram = gl.createProgram();
8.   gl.attachShader(shaderProgram, vertexShader);
9.   gl.attachShader(shaderProgram, fragmentShader);
10.  gl.linkProgram(shaderProgram);
```

```
11.
12.  // If creating the shader program failed, alert
13.
14.  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {

15.      alert("Unable to initialize the shader program.");
16.  }
17.
18.  gl.useProgram(shaderProgram);
19.
20.  vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
    "aVertexPosition");
21.  gl.enableVertexAttribArray(vertexPositionAttribute);
22. }
```

Funkcija učitava dva programa za sjenčanje. Prvi je program za sjenčanje fragmenata (eng. *fragment shader* ili *pixel shader*), koji se učitava iz skriptnog elementa s identifikacijskim imenom „shader-fs“. Drugi je program za sjenčanje vrhova (eng. *vertex shader)*, koji se učitava iz elementa „shader-vs“. `getShader()` funkcija uzima programe za sjenčanje iz DOM-a.

Kreiramo program za sjenčanje pozivom funckije WebGL objekta `createProgram()`, kojemu pričvršćujemo dva programa za sjenčanje i vežemo ih. Nakon toga parametar `LINK_STATUS` gl objekta se provjerava kako bi se vidjelo da li su programi uspješno vezani, te ako da aktiviramo novi program za sjenčanje.

3.2.2 Učitavanje programa za sjenčanje iz DOM-a

Funkcija `getShader()` dohvaća program za sjenčanje sa specificiranim imenom iz DOM-a, te vraća prevedeni program za sjenčanje ili null ako nije mogao biti učitan ili preveden.

```
1. function getShader(gl, id) {
2.     var shaderScript, theSource, currentChild, shader;
3.
4.     shaderScript = document.getElementById(id);
5.
6.     if (!shaderScript) {
7.         return null;
8.     }
9.
10.    theSource = "";
11.    currentChild = shaderScript.firstChild;
12.
13.    while(currentChild) {
14.    if (currentChild.nodeType == currentChild.TEXT_NODE) {
15.        theSource += currentChild.textContent;
16.    }
```

```
17.
18.     currentChild = currentChild.nextSibling;
19.   }
```

Kada se element sa specificiranim ID-em nađe, njegov se tekst čita u varijalbu `theSource`.

```
1. if (shaderScript.type == "x-shader/x-fragment") {
2.     shader = gl.createShader(gl.FRAGMENT_SHADER);
3.   } else if (shaderScript.type == "x-shader/x-vertex") {
4.     shader = gl.createShader(gl.VERTEX_SHADER);
5.   } else {
6.      // Unknown shader type
7.      return null;
8.   }
```

Kada se pročita kod programa za sjenčanje, pogledamo MIME tip objekta za sjenčanje kako bi odredili da li je program za sjenčanje vrhova (MIME type "x-shader/x-vertex") ili program za sjenčanje fragmenata (MIME type "x-shader/x-fragment") te kreiramo određeni tip programa za sjenčanje iz dohvaćenog koda.

```
1. gl.shaderSource(shader, theSource);
2.
3.   // Compile the shader program
4.   gl.compileShader(shader);
5.
6.   // See if it compiled successfully
7.   if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
8.       alert("An error occurred compiling the shaders: " + gl.get
   ShaderInfoLog(shader));
9.       return null;
10.        }
11.
12.  return shader;
```

Konačno, izvorni kod je proslijeđen programu za sjenčanje i preveden. Ako dođe do pogreške kod prevođenja, vraća se null. U protivnom novo-prevedeni program za sjenčanje se vraća u pozivajući program.

3.2.3 Programi za sjenčanje

Moramo dodati same programe za sjenčanje u naš HTML dokument.

3.2.3.1 *Program za sjenčanje fragmenata*

Svaki slikovni element u poligonu se zove fragment u GL jeziku. Posao programa za sjenčanje fragmenata je uspostaviti boju svakog slikovnog elementa. U ovom slučaju dodjeljujemo bijelu boju svakom slikovnom elementu.

gl_FragColor je ugrađena GL varijabla koja se koristi za boju fragmenata. Postavljanje vrijednosti u varijablu određuje boju slikovnog elementa.

```
<script id="shader-fs" type="x-shader/x-fragment">
  void main(void) {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
  }
</script>
```

3.2.3.2 *Programi za sjenčanje vrhova*

Program za sjenčanje vrhova *(eng. vertex shader)* definira poziciju svakog vrha tako da projicira koordinate geometrije u koordinate zaslona.

```
 <script id="shader-vs" type="x-shader/x-vertex">
   attribute vec3 aVertexPosition;

   uniform mat4 uMVMatrix;
   uniform mat4 uPMatrix;

   void main(void) {
gl_Position = uPMatrix*uMVMatrix*vec4(aVertexPosition, 1.0);
   }
</script>
```

3.3 Kreiranje objekta

Prije nego što možemo nacrtati objekt, moramo kreirati spremnik koji sadrži vrhove. To radimo s funkcijom `initBuffers()`.

```
var horizAspect = 480.0/640.0;

function initBuffers() {
  squareVerticesBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, squareVerticesBuffer);

  var vertices = [
    1.0,  1.0,  0.0,
   -1.0, 1.0,  0.0,
    1.0,  -1.0, 0.0,
   -1.0, -1.0, 0.0
  ];

```

```
14. gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.
    STATIC_DRAW);
15. }
```

Funkcija započinje pozivom metode `createBuffer()` gl objekta, koja dohvaća spremnik u koji se kasnije spremaju vrhovi. Spremnik se kasnije veže uz kontekst pozivom metode `bindBuffer()`.

Nakon toga kreiramo JavaScript niz koji sadrži koordinate za svaki vrh koji želimo nacrtati. Niz se pretvara u niz WebGL float brojeve, te se stavlja u gl objekt `bufferData()` metodu.

3.4 Crtanje scene

Nakon što su programi za sjenčanje postavljeni i objekt je konstruiran, možemo nacrtati scenu. Kako se u primjeru ne animira, `drawScene()` funkcija je dosta jednostavna.

```
1. function drawScene() {
2.  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
3.
4.  perspectiveMatrix = makePerspective(45, 640.0/480.0, 0.1, 100.0
    );
5.
6.  loadIdentity();
7.  mvTranslate([-0.0, 0.0, -6.0]);
8.
9.  gl.bindBuffer(gl.ARRAY_BUFFER, squareVerticesBuffer);
10. gl.vertexAttribPointer(vertexPositionAttribute, 3, gl.FLOAT,
      false, 0, 0);
11. setMatrixUniforms();
12. gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
13. }
```

Prvo korak je očistiti kontekst u boju pozadine, zatim odredimo perspektivnu kamere. Postavimo vidno polje od 45° s omjerom širine i visine 640/480 (dimenzije canvasa). Također odredimo da želimo nacrtati predmete između 0.1 i 100 jedinica.

Zatim postavimo poziciju objekta tako da učitamo poziciju identiteta koju translatiramo od kamere za 6 jedinica, nakon čega vežemo spremnik vrhova na kontekst, konfiguriramo ga i crtamo objekt pozivom `drawArrays()` metode.

3.5 Primjena boje na vrhovima

U GL-u, objekti se grade koristeći skupove vrhova, od kojih svaki ima poziciju i boju. Po definiciji, boje drugih slikovnih elemenata (i drugi atributi, uključujući pozicije) su računate linearnom interpolacijom, čime se automatski kreira glatki gradijent. U prošlom primjeru program za sjenčanje vrhova nije primijenio specifične boje vrhovima. Uz to program za sjenčanje fragmenata je zadao fiksnu boju (bijelu) svakom slikovnom elementu, pa je tako rezultat bio cijeli bijeli kvadrat.

Kada bi htjeli nacrtati gradijent kod kojeg je svaki kut kvadrata drugačije boje (crvena, plava, zelena, bijela) prvo što trebamo napraviti je uspostaviti boje svakog vrha. Kako bi to napravili moramo kreirati niz boja vrhova koje ćemo spremiti u WebGL spremnik.

```
var colors = [
    1.0,  1.0,  1.0,  1.0,    // white
    1.0,  0.0,  0.0,  1.0,    // red
    0.0,  1.0,  0.0,  1.0,    // green
    0.0,  0.0,  1.0,  1.0     // blue
  ];

  squareVerticesColorBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, squareVerticesColorBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
```

Alociramo novi WebGL spremnik koji će spremiti boje, te boje pretvaramo u WebGL float brojeve i spremamo ih u spremnik.

Kako bi se boje koristile, treba ažurirati program za sjenčanje vrhova s prikladnim bojama iz spremnika boja.

```
<script id="shader-vs" type="x-shader/x-vertex">
      attribute vec3 aVertexPosition;
      attribute vec4 aVertexColor;

      uniform mat4 uMVMatrix;
      uniform mat4 uPMatrix;

      varying lowp vec4 vColor;

     void main(void) {
     gl_Position = uPMatrix*uMVMatrix*vec4(aVertexPosition, 1.0);
     vColor = aVertexColor;
     }
</script>
```

Ključna razlika ovdje je da za svaki vrh postavljamo odgovarajuću vrijednost iz niza boja.

3.5.1 Bojanje fragmenata

Kako bi uzeli interpoliranu boju za svaki slikovni element, jednostavno moramo uzimati vrijednosti iz varijable `vColor`:

```
<script id="shader-fs" type="x-shader/x-fragment">
        varying lowp vec4 vColor;

      void main(void) {
        gl_FragColor = vColor;
      }
</script>
```

Svaki fragment jednostavno primi interpoliranu boju temeljenu na svojoj poziciji određenoj relativno prema vrhovima, umjesto fiksne vrijednosti.

3.5.2 Crtanje koristeći boje

Potrebno je dodati kod u `initShaders()` funkciji kako bi inicijalizirali atribut boje za program za sjenčanje.

```
vertexColorAttribute = gl.getAttribLocation(shaderProgram, "aVertexColor");
gl.enableVertexAttribArray(vertexColorAttribute);
```

Nakon toga, `drawScene()` možemo ispraviti kako bi se koristile boje kada crtamo objekt (u primjeru kvadrat).

```
gl.bindBuffer(gl.ARRAY_BUFFER, squareVerticesColorBuffer);
gl.vertexAttribPointer(vertexColorAttribute, 4, gl.FLOAT, false, 0, 0);
```

Rezultat (slika 23.):

Slika 23. Primjer bojanja

3.6 Three.js

Three.js je popularna biblioteka za WebGL koja apstrahira rad u 3D u preglednicima. Omogućuje kreiranje kamera, objekata, svijetla, materijala i još puno toga uz izbor načina crtanja scene. Tako se može birati između crtanja scene koristeći HTML5 canvas, WebGl ili SVG. Standardni izbori su canvas ili WebGL. Canvas je šire podržan od WebGL-a, ali WebGL radi za GPU grafičke kartice, što znači da se CPU može koncentrirati na druge zadatke kao što su korisnička interakcija [21].

3.6.1 Scena

Prvi korak je postavljanje `div` elementa koji će sadržavati produkt (crtež), zatim na jednostavan način kreiramo scenu, dodajemo kameru u scenu, te instanciramo i pokrećemo program za ostvarivanje prikaza.

- *Renderer*: `new THREE.WebGLRenderer()`
- Kreiranje kamere: `new THREE.PerspectiveCamera(fov, aR, n, f)`
- Kreiranje scene: `new THREE.Scene()`
- Kreiranje mreže sa geometrijom i materijalima:
  ```
  new THREE.Mesh(new THREE.CubeGeometry(w,h,d),
  new THREE.MeshBasicMaterial({color: 0x000000}))
  ```
- Dodavanje mreže u scenu: `.add(mesh)`
- Kreiranje prikaza scene: `renderer.render(scene, camera)`
- Dodavanje *renderera* u web stranicu:
  `document.body.appendChild(renderer.domElement)`

Jednostavan primjer kako to izgleda u praksi:

```
1. // veličina scene
2. var WIDTH = 400,
3.     HEIGHT = 300;
4.
5. // set some camera attributes
6. var VIEW_ANGLE = 45,
7.     ASPECT = WIDTH / HEIGHT,
8.     NEAR = 0.1,
9.     FAR = 10000;
10.
11.  // postavljanje div elementa u dokumenta
12.  container = document.createElement( 'div' );
13.  document.body.appendChild( container );
14.
15.  // kreiranje scene, kamere i renderer
16.  var scene = new THREE.Scene();
17.
18.  var camera = new THREE.PerspectiveCamera(
19.                    VIEW_ANGLE,
20.                    ASPECT,
21.                    NEAR,
22.                    FAR );
23.
24.  // početna pozicija kamere je 0,0,0 pa ju moramo pomaknuti
25.  camera.position.z = 300;
26.  scene.add( camera );
27.
28.  var renderer = new THREE.WebGLRenderer();
29.
30.  // pokrećemo renderer
31.  renderer.setSize(WIDTH, HEIGHT);
32.
33.  // dodajemo render-dobiveni DOM element
34.  container. appendChild (renderer.domElement);
```

3.6.2 Stvaranje mreže

Nakon što smo postavili scenu, kameru i način crtanja, moramo odrediti što ćemo crtati. Three.js dolazi uz potporu za učitavanje nekoliko različitih standardnih tipova, no za početni primjer koristiti ćemo primitive (geometrijske mreže) kao što su sfere, ravnine, kocke i cilindri koje možemo na jednostavan način kreirati pomoću Three.js.

```
1. // parametri sfere
2. var radius = 50, segments = 16, rings = 16;
3.
4. // kreiranje nove mreže sa geometrijom sfere
5. var sphere = new THREE.Mesh(
6. new THREE.SphereGeometry(radius,
7. segments,
```

```
8. rings),
9.
10.  sphereMaterial);
11.
12.  // stavljanje sfere u scenu
13.  scene.add(sphere);
```

Ovdje još nije definiran materijal sfere (varijabla `sphereMaterial`).

3.6.3 Materijali

Jedan od korisnijih dijelova Three.js je način primijene materijala. Biblioteka osigurava nekoliko uobičajenih i vrlo korisnih materijala za primjenu nad mrežom:

- Osnovna – crta „unlit“
- Lambert
- Phong i dr.

Ovo je izrazito korisno jer je u WebGL-u potrebno pisanje programa za sjenčanje za sve što se crta. To znači da se mora oponašati matematika osvjetljavanja, relfeksije itd. Tako stvari brzo postaju vrlo komplicirane. Three.js to apstrahira i olakšava. Ako je pak potrebno ili se želi pisati vlastiti program za sjenčanje, i to se može s `MeshShaderMaterial` metodom, pa tako biblioteka daje fleksibilnost.

```
1. // kreiranje materijala scene
2. var sphereMaterial = new THREE.MeshLambertMaterial(
3. {
4.     color: 0xCC0000
5. });
```

Boja je samo jedna od svojstva koja se mogu specificirati kada se kreira materijal.

3.6.4 Svjetla

Kada bi se sada crtala scena, vidjeli bi samo crveni krug. Bez postavljene svjetlosti unutar scene, Three.js po definiciji postavlja puno prostorno osvjetljenje što je isto kao konstantno sjenčanje. Postavljamo jednostavnu točku svjetlosti:

```
1. // kreiranje točkastog svijetla
2. var pointLight = new THREE.PointLight( 0xFFFFFF );
3.
4. // postavljanje pozicije
5. pointLight.position.x = 10;
6. pointLight.position.y = 50;
7. pointLight.position.z = 130;
```

```
8.
9. // dodavanje u scenu
10. scene.add(pointLight);
```

3.6.5 Sjene

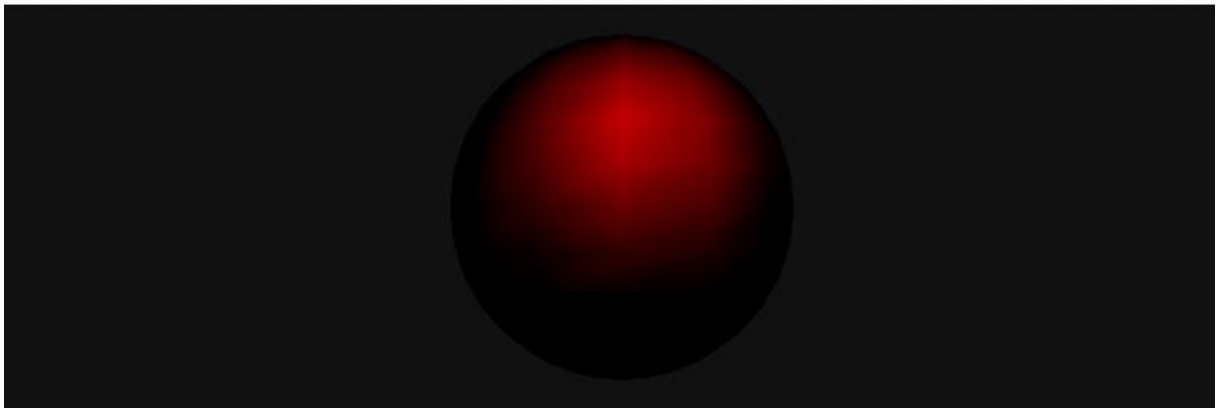
Three.js ima mape sjena (eng. *shadow maps*), te jedino što je potrebno je njihovo omogućavanje za pojedino svijetlo i objekt. Sjene rade samo za *Spotlight* vrstu svijetla.

```
1.   // enable shadows on the renderer
2.   renderer.shadowMapEnabled = true;
3.
4.   // enable shadows for a light
5.   light.castShadow = true;
6.
7.   // enable shadows for an object
8.   litCube.castShadow = true;
9.   litCube.receiveShadow = true;
```

3.6.6 Crtanje scene

Postavili smo sve potrebno za crtanje, pa sada možemo vidjeti rezultat:

```
10.        // crtanje!
11.        renderer.render(scene, camera);
```



Slika 24. Primjer crtanja sfere s WebGL

3.6.7 Opća svojstva objekta

Svi objekti nasljeđuju svojstva od nadklase `Object3D` koji sadrži neka vrlo korisna svojstva kao što su `position` (pozicija), `rotation` (rotacija) i `scale` (skaliranje) informacije. U našem primjeru sfera je `Mesh` koji nasljeđuje Object3D i kojemu dodaje svojstva: `geometry` i `materials`.

```
1.  // geometrija
2.  sphere.geometry
3.
4.  // koja sadrži vrhove i ravnine
5.  sphere.geometry.vertices // array
6.  sphere.geometry.faces // array
7.
8.  // pozicija
9.  sphere.position // x, y, z
10. sphere.rotation // x, y, z
11. sphere.scale // x, y, z
```

3.6.8 Odabir objekata

Kako bi mogli interaktivno raditi s objektima, moramo ih nekako odabrati da znamo s kojim rukujemo (eng. *select*). Kako to možemo napraviti? Pritiskom miša projektiramo zraku u scenu i pronalazimo sjecišta sa objektima.

```
1. var projector = new THREE.Projector();
2. window.addEventListener('mousedown', function (ev){
3.  if (ev.target == renderer.domElement) {
4.   var x = ev.clientX;
5.   var y = ev.clientY;
6.   var v = new THREE.Vector3((x/width)*2-1, -(y/height)*2+1,
   0.5);
7.   projector.unprojectVector(v, camera);
8.   var ray = new THREE.Ray(camera.position,
9.                 v.subSelf(camera.position).normalize());
10.  var intersects = ray.intersectObjects(controller.objects);
11.  if (intersects.length > 0) {
12.        controller.setCurrent(intersects[0].object);
13.   }
14.  }
15. }, false);
```

4 Zaključak

Uvođenje WebGL-a otvara vrata mogućnosti naprednoj web grafici bez potrebe priključaka. Web programeri više nisu ograničeni na korištenje samo slika ili alternativnih rješenja kao što je Flash. Jedino što je potrebno posjetiocima stranica je moderan web preglednik koji podržava WebGL.

Iako Canvas i WebGL imaju svojih sličnosti, WebGL je podosta drugačiji od canvas tehnike crtanja. Osnovno poznavanje 2D grafike i JavaScripta je dovoljno kako bi se moglo skočiti u <canvas> vode. Canvas osigurava jednostavno sučelje pomoću kojih se može doći do jednostavnih rezultata vrlo brzo.

WebGL je puno moćniji, ali zahtijeva više učenja, osobito za one koji su se bavili samo web programiranjem. Za one koji već znaju OpenGL ES vrlo dobro WebGL je jednostavan za naučiti.

Veliki dio WebGL aplikacije je napisan u GLSL-u i to je najveća razlika WebGL-a prema canvasu. On je jezik niske razine, sličan programskom jeziku C, koji je napravljen za brzo izvođenje na modernim GPU-ima, kako bi se što više iskoristila sklopovska funkcionalnost bez ograničavanja prenosivosti. U pogledu ciljeva dizajna, potpuno je suprotan od JavaScripta, koji je jezik vrlo visoke razine i žrtvuje učinkovitost u korist apstrakcije [1].

Broj JavaScript biblioteka koje osiguravaju funkcionalnosti visoke razine za kreiranje 3D grafičkih aplikacija raste. Trenutno je po istraživanju autora ove radnje najpopularniji Three.js. Three.js je vrlo jednostavan za korištenje, a samo neke stvari koje osigurava su *renderer* za <canvas>, <svg> i WebGL, dodavanje i micanje objekata u sceni u vremenu izvođenja, postavljanje ortografske i perspektivne kamere, praćenje njezinih kontrola, animacija, svijetla, materijali objekata, programa za sjenčanje, i još puno toga.

5 Literatura

1. *Web Graphics by David Chisnall*, 2012, http://www.informit.com/artcles/article.aspx?p=1609154, (pristupljeno 15.04.2012.).

2. *Graphics - HTML5 Rocks,* http://www.html5rocks.com/en/features/graphics, (pristupljeno 15.04.2012.).

3. *Developer Opera – HTML5 canvas- the basics,* 08.01.2009., http://dev.opera.com/articles/view/html-5-canvas-the-basics/.(pristupljeno 18.04. 2012.)

4. *Wikipedia – SVG,* 01.03.2012., http://en.wikipedia.org/wiki/Scalable_Vector_Graphics. (pristupljeno 15.04.2012.)

5. *Wikipedia – WebGL,* 02.04.2012., http://en.wikipedia.org/wiki/WebGL. (pristupljeno 18.04 2012.)

6. Addison-Wesley, Learning HTML5 Game Programming, A Hands-on Guide to Building Online Games Using Canvas SVG and WebGL. 2012.

7. Developer Mozilla – *Canvas tutorial* – Basic usage, 09.10.2008., https://developer.mozilla.org/en/Canvas_tutorial%3ABasic_usage, (pristupljeno 25.04.2012.)

8. Developer Mozilla – *Drawing Graphics with Canvas*, 12.08.2011., https://developer.mozilla.org/en/Drawing_Graphics_with_Canvas, (pristupljeno 25.04.2012.).

9. KinetisJS, http://www.kineticjs.com, (pristupljeno 25.04.2012.).

10. Rowell, Eric: HTML5 *Canvas KineticJS Rect Tutorial*, 24.02.2012., http://www.html5canvastutorials.com/kineticjs/html5-canvas-kineticjs-rect-tutorial/, (pristupljeno 25.04.2012.).

11. Rowell, Eric: *HTML5 Canvas Shape Events with KineticJS,* 16.04.2011., http://www.html5canvastutorials.com/kineticjs/html5-canvas-path-mouseover/, (pristupljeno 01.05.2012.).

12. Rowell, Eric: *HTML5 Canvas Multi-Event Binding Tutorial,* 02.01.2012., http://www.html5canvastutorials.com/kineticjs/html5-canvas-multi-event-binding/, (pristupljeno 02.05.2012.).

13. Rowell, Eric: *HTML5 Canvas Pixel Detection with KineticJS*, 15.04.2012., http://www.html5canvastutorials.com/kineticjs/html5-canvas-pixel-detection-with-kineticjs/, (pristupljeno 02.05.2012.).

14. Rowell, Eric: *HTML5 Canvas Drag and Drop Tutorial,* 17.01.2011., http://www.html5canvastutorials.com/kineticjs/html5-canvas-drag-and-drop-tutorial/, (pristupljeno 05.05.2012.).

15. Rowell, Eric: *HTML5 Canvas Shape Layering with KineticJS*, 22.01.2012., http://www.html5canvastutorials.com/kineticjs/html5-canvas-shape-layering-with-kineticjs/, (pristupljeno 05.05 2012.).

16. *Rowell, Eric: HTML5 Canvas Linear Transition Tutorial with KineticJS,* 18.03.2012., http://www.html5canvastutorials.com/kineticjs/html5-canvas-linear-transition-tutorial-with-kineticjs/, (pristupljeno 06.05.2012.).

17. Rowell, Eric: *HTML5 Canvas All Easing Functions with KineticJS,* 09.04.2012., http://www.html5canvastutorials.com/kineticjs/html5-canvas-all-easing-functions-with-kineticjs/, (pristupljeno 06.05.2012.).

18. KHRONOS WebGL wiki, 10.04.2011., http://www.khronos.org/webgl/wiki/Getting_Started, (pristupljeno 06.05.2012.).

19. Developer Mozilla – *Getting started with WebGL*, 07.04.2012., https://developer.mozilla.org/en/WebGL/Getting_started_with_WebGL, (pristupljeno 05.05.2012.).

20. Developer Mozilla – *Adding 2d content to a WebGL context,* 02.05.2012., https://developer.mozilla.org/en/WebGL/Adding_2D_content_to_a_WebGL_context, (pristupljeno 06.05.2012.).

21. Lewis, Paul: *Getting started with Three.js* - HTML5 Rocks, 02.06.2011., http://www.html5rocks.com/en/tutorials/three/intro/, (pristupljeno 10.04.2012.).

22. *Three.js*, 2012., http://mrdoob.github.com/three.js/, (pristupljeno u svibnju 2012.).

6 Sažetak

Naslov: **Interaktivni grafički objekti na web-u**

U radu su proučene najnovije tehnologije za ostvarivanje prikaza grafike na web-u. Dan je kratak pregled osnova korištenja canvasa (2D grafika) i JavaScript biblioteke KineticJS koja proširuje 2D kontekst omogućujući canvas interaktivnost za desktop i mobilne aplikacije. Opisan je uvod u WebGL koji omogućava prikaz 3D grafike unutar bilo kojeg kompatibilnog web preglednika bez korištenja dodatnih priključaka. Obrađene su osnove korištenja JavaScript biblioteke Three.js koja olakšava rad u WebGL-u. Na različitim primjerima pokazane su osnove u korištenju navedenih tehnologija.

Ključne riječi: web grafika, Canvas, WebGL, KineticJS, Three.js

7 Abstract

Title: **Interactive graphic objects on the Web**

The newest web graphics technologies have been researched in this work. A short review of the basic use of canvas and its JavaScript library KineticJS, that extends the 2D context by enabling interactivity for desktop and mobile applications, is given. This work will introduce you to the basics of using WebGL, a JavaScript API for rendering interactive 3D graphics within any compatible web browser without the use of plug-ins. Additionally to WebGL, a short introduction to Three.js, a JavaScript library that facilitates work in WebGL is elaborated. The basic use of the given technologies are elaborated on different examples.

Key words: web graphics, Canvas, WebGL, KineticJS, Three.js