SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 615

2D FIZIKALNI POGON

Luka Samac

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 615

2D FIZIKALNI POGON

Luka Samac

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

ZAVRŠNI ZADATAK br. 615

Pristupnik:	**Luka Samac (0036524776)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **2D Fizikalni pogon**

Opis zadatka:

Proučiti osnove fizikalnog ponašanja objekata. Proučiti osnove izrade fizikalnog pogona. Razraditi koncepte fizikalnog pogona u dvije dimenzije na jednostavnim likovima. Ostvariti implementaciju dvodimenzionalnog fizikalnog pogona. Načiniti testiranje na nizu primjera. Analizirati i ocijeniti ostvarene rezultate. Diskutirati upotrebljivost ostvarenih rezultata kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 10. lipnja 2022.

Sadržaj

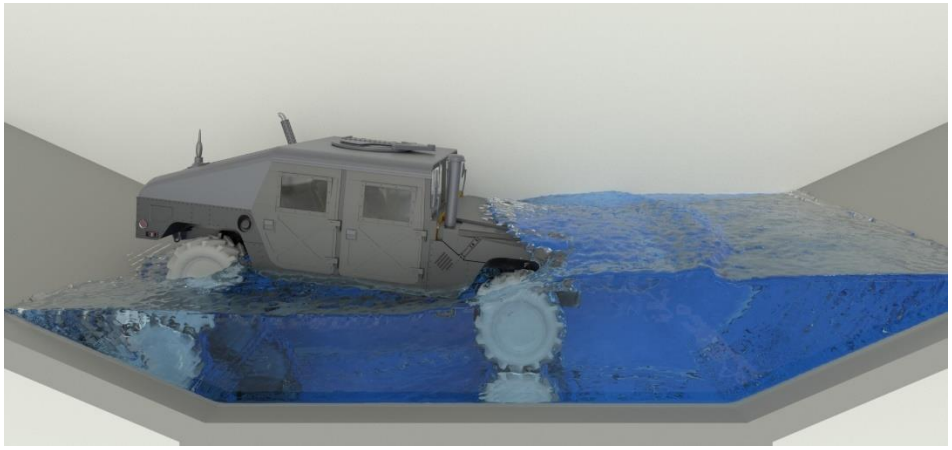
Uvod

Od začetka računalne grafike, ljudi su pokušavali prikazivati razne fizikalne pojave na računalima. Fizikalni pogoni imaju razne svrhe, mogu služiti za razne testove, za prikazivanje fizikalnih pojava u edukacijske svrhe te su čest gost u videoigrama.

Bez obzira na svrhu, fizikalni pogoni kombiniraju razna znanja iz područja fizike i računalne znanosti. Već od 80-ih godina, ljudi su simulirali pojave poput dinamike fluida na superračunalima.

Potreba za fizikalnim pogonima je rasla tijekom godina te su fizikalni pogoni se podijelili na dvije vrste:

- Fizikalne pogone visoke preciznosti koji se koriste u znanosti te raznim drugim područjima
- Fizikalni pogoni u stvarnom vremenu koji se često koriste u videoigrama, u edukacijske svrhe te mnoge druge (Slika 0.1)



Slika 0.1 Project Chrono, fizikalni pogon u stvarnom vremenu

U sklopu ovog rada, proučit će se faze izrade fizikalnog pogona te fizikalna podloga potrebna za njegovu izradu. Tema simulacije su kruta tijela i sudari, a implementacija je napravljena po uzoru na osnovne koncepte fizikalnog pogona matter.js [7] u jeziku Javascript.

1. Osnove 2D fizikalnog pogona

Prilikom izrade 2D fizikalnog pogona, potrebno je odabrati što točno želimo simulirati te kakvu vrstu pogona želimo izraditi.

Fizikalni pogoni se dijele na [1]:

- pogone visoke preciznosti
- pogone u stvarnom vremenu.
  - statički pogoni
  - dinamički pogoni

Pogone visoke preciznosti koristimo kada želimo simulirati fizikalne pojave koje su teške za izračunati te je za simulaciju potrebno više vremena. Za većinu ostalih slučajeva koristimo pogone u stvarnom vremenu. Statički pogoni u stvarnom vremenu očekuju da točno određeno vrijeme prođe između svakog iscrtavanja, dok dinamički pogoni dobiju vrijeme koje je prošlo između dva iscrtavanja okvira te ga koriste za računanje [1].

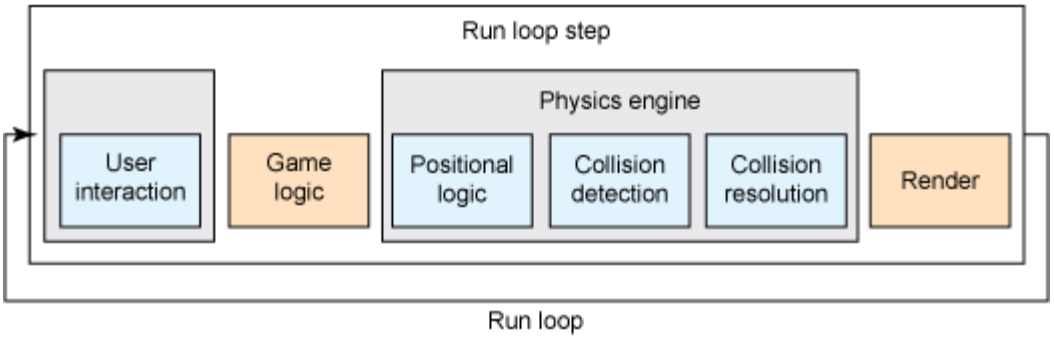
Odabir dijela fizike kojeg želimo simulirati je vrlo bitno jer time odabiremo koje formule ćemo koristiti pri izradi pogona. Kompleksnost fizikalnog pogona jako ovisi o ovom odabiru. Fizika krutih tijela je čest odabir pri izradi fizikalnih pogona jer je deformacija tijela zanemariva te je udaljenost bilo koje dvije točke na tijelu konstantna bez obzira na vanjsku silu koja na njega djeluje.

Za potrebe ovog rada je dovoljno napraviti statički pogon u stvarnom vremenu koji implementira fiziku krutih tijela.

1.1. Petlja fizikalnog pogona

Petlja fizikalnog pogona je jezgra cijelog sustava. Iako ovaj dio zahtijeva relativno malo koda, vrlo je važno na koji način implementiramo ovu petlju. Krivi poredak u ovoj petlji može utjecati na ispravnosti rada ili performanse.

Primjer izgleda ovakve petlje je vidljivo na slici (Slika 1.1).



Slika 1.1 Petlja fizikalnog pogona

U prvom koraku potrebno je napraviti svu interakciju s korisnikom što većinom dolazi putem miša i tipkovnice. To može biti dodavanje, pomicanje, brisanje objekata, mijenjanje svojstva objekata te mnoge druge mogućnosti.

Nakon toga potrebno je napraviti svu logiku koja nije dio samog fizikalnog pogona. Ovaj dio nije potrebno implementirati u ovom radu, ali je čest korak ako se pogon koristi u npr. videoigrama.

U trećem koraku računaju se sve stvari potrebne za simulaciju fizike objekata. Ovaj korak se dijeli na 3 dijela [1]:

- pozicijska logika
- detekcija sudara
- rezolucija sudara

U dijelu pozicijske logike, računa se nova pozicija objekta ovisno o njegovoj brzini i akceleraciji. Nakon izračuna nove pozicije svih objekata, mogu se izračunati sudari između tijela u sceni te napraviti rezolucija svih sudara.

U zadnjem koraku radi se iscrtavanje svih objekata na sceni.

Ponavljanje petlje može se obaviti putem funkcije *requestAnimationFrame(mainLoop)* u jeziku Javascript. Korištenje ove funkcije omogućuje da se ne troše računalni resursi kada korisnik nije na istom prozoru u pregledniku. Ovo je glavni razlog zašto koristiti ovu funkciju umjesto alternativa poput funkcije *setInterval()*. U slučaju da želimo napraviti da broj okvira u sekundi bude konstantan za funkciju *requestAnimationFrame()* (npr. 60 okvira u sekundi), moramo napraviti svoju implementaciju koja će ograničiti broj okvira ili pronaći već postojeće implementacije na Internetu.

1.2. Fizikalna svojstva

Svaki objekt u sceni ima niz svojstava koja su mu potrebna za simulaciju fizike i iscrtavanje. Odabir svojstava ovisi o potrebama simulacije te kompleksnosti kalkulacija.

Za izračun pozicijske logike, potrebno je za svaki objekt imati trenutnu poziciju, brzinu koja će utjecati na promjenu pozicije te akceleraciju koja će utjecati na promjenu brzine. Na ove vrijednosti mogu utjecati razni faktori poput trenja koje će smanjivati brzinu objekta kroz vrijeme.

Dodatno, moramo modelirati i kut objekta kako bi se mogao rotirati. Kut također ima svoju vrijednost te pripadnu brzinu i akceleraciju.

Za simulaciju krutih tijela, nije potrebno imati ove vrijednosti za svaku točku objekta jer se sve točke uvijek pomiču istom brzinom i na istoj su udaljenosti jedna od druge. Zbog toga je dovoljno imati samo jednu vrijednost po svojstvu za cijeli objekt.

Za rezoluciju sudara potrebno nam je imati dodatna svojstva poput mase i momenta inercije. Rezolucija te izračunava putem fizikalnih formula za sudar krutih tijela.

1.3. Računanje s vektorima i matricama

Za računanje fizikalnih pojava, čest je slučaj da je potrebno računati s vektorima i matricama. Javascript nema implementaciju ni vektora ni matrica, ali postoje mnoge biblioteke koje omogućuju rad s njima.

Vektori su korišteni za skoro sav izračun fizike u implementaciji dok su matrice većinom korištene za izračun nove pozicije točaka nakon rotacije za određeni kut.

Za simulaciju 2D krutih tijela, nisu potrebno kompleksne operacije s vektorima i matricama te je zato korištena vlastita implementacije koji sadrži osnovne operacije s njima.

```
class Vector{
    constructor(x, y){
        this.x = x;
        this.y = y;
    }
    set(x, y){
        this.x = x;
```

```
        this.y = y;
    }
    add(v){
        return new Vector(this.x+v.x, this.y+v.y);
    }
    subtr(v){
        return new Vector(this.x-v.x, this.y-v.y);
    }
    mag(){
        return Math.sqrt(this.x**2 + this.y**2);
    }
    mult(n){
        return new Vector(this.x*n, this.y*n);
    }
    normal(){
        return new Vector(-this.y, this.x).unit();
    }
    unit(){
        if(this.mag() === 0){
            return new Vector(0,0);
        } else {
            return new Vector(this.x/this.mag(),
this.y/this.mag());
        }
    }
    static dot(v1, v2){
        return v1.x*v2.x + v1.y*v2.y;
    }
    static cross(v1, v2){
        return v1.x*v2.y - v1.y*v2.x;
    }
}
```

Programski kod 1.1 – Implementacija 2D vektora

```
lass Matrix{
    constructor(rows, cols){
        this.rows = rows;
        this.cols = cols;
```

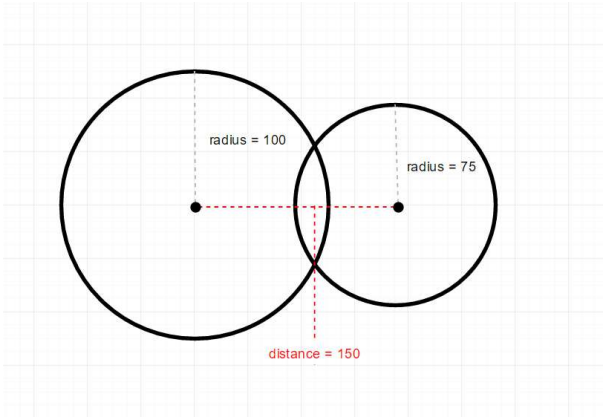
```
        this.data = [];
        for (let i = 0; i<this.rows; i++){
            this.data[i] = [];
            for (let j=0; j<this.cols; j++){
                this.data[i][j] = 0;
            }
        }
    }
    multiplyVec(vec){
        let result = new Vector(0,0);
        result.x = this.data[0][0]*vec.x +
this.data[0][1]*vec.y;
        result.y = this.data[1][0]*vec.x +
this.data[1][1]*vec.y;
        return result;
    }
    rotMx22(angle){
        this.data[0][0] = Math.cos(angle);
        this.data[0][1] = -Math.sin(angle);
        this.data[1][0] = Math.sin(angle);
        this.data[1][1] = Math.cos(angle);
    }
}
```

Programski kod 1.2 – Implementacija 2x2 matrica

2. Detekcija sudara

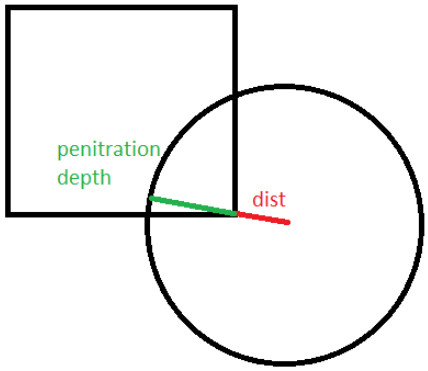
Detekcija sudara vrlo je široko područje s puno raznih algoritama koji su primjenjivi za određene slučajeve te raznih kompleksnosti i performansi. Cilj je imati algoritam koji je što manje kompleksan te ima što bolje performanse, a da zadovoljava sve slučajeve potrebne za simulaciju.

Jednostavan primjer detekcije sudara bio bi sudar za dva kruga (Slika 2.1). U ovom slučaju jedino je potrebno provjeriti je li udaljenost središta kruga manje od zbroja radijusa. Ovaj izračun potrebno je ponoviti za svaki par krugova u sceni.



Slika 2.1 Detekcija sudara s dva kruga

Stvari postaju teže kada dodamo novu vrstu objekata poput pravokutnika. Čak iako implementiramo detekciju sudara između dva pravokutnika, kako detektirati sudar između pravokutnika i kruga (Slika 2.2)? U slučaju još novih vrsta objekata, bilo bi potrebno imati definiranu detekciju sudara sa svim starim vrstama objekta.



Slika 2.2 Detekcija sudara s pravokutnikom i krugom

Zbog ovakvih problema potrebno je imati detekciju sudara koju je moguće lagano skalirati s količinom raznovrsnih objekata. Ovo nije lagan problem za riješiti te ovisi o vrsti objekata koji se mogu pojaviti u simulaciji te o samom algoritmu za detekciju.

Kako bi se olakšalo računanje detekcije sudara, u implementaciji su definirani primitivni oblici na kojima će se računati kolizija. Ti oblici su:

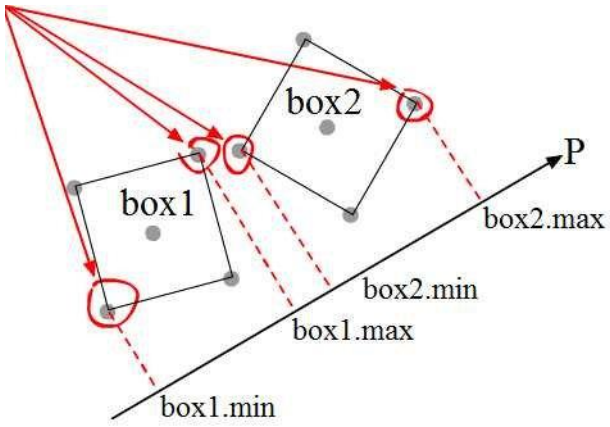
- linija
- krug
- pravokutnik
- trokut

Svi oblici koji se pojavljuju u sceni su skup primarnih oblika (npr. kapsula je pravokutnik s dva kruga na rubovima). Sudar je tada potrebno računati na način da se provjeri sudar na svim parovima primarnih objekata između dva tijela. Ako prvi objekt ima 2 primarna oblika, a drugi ima 3 oblika, tada će se izvršiti ukupno 6 detekcija sudara između ta dva objekta. U slučaju da je pronađeno više sudara za dva objekta, potrebno je odabrati najpovoljniji sudar.

2.1. SAT algoritam

SAT (eng. *Separating Axis Theorem*) algoritam vrlo je popularna metoda za detekciju sudara konveksnih objekata [3]. Teorem tvrdi da za svaka dva konveksna objekta koji se ne sudaraju postoji os projekcije u kojoj ti objekti nisu u sudaru [4]. Način odabira ovih osi te njihov broj ovisi o obliku objekata za koji provjeravamo sudar.

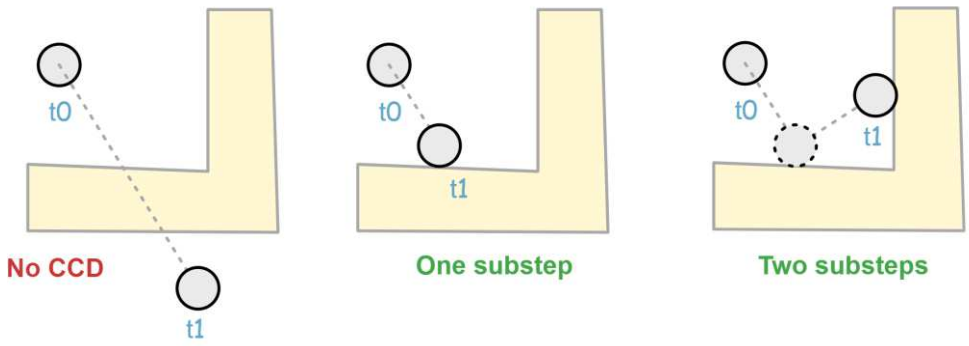
Svi primitivni oblici u implementaciji su konveksni te je SAT algoritam primjenjiv za ovaj rad. Potrebno je definirati osi koje se traže za svaku vrstu primitivnog oblika te izračunati projekciju svih točaka dvaju objekata na tu os. Kada su točke projicirane na os, potrebno je samo gledati minimum i maksimum projekcije za svaki objekt (Slika 2.3).



Slika 2.3 SAT algoritam s dva pravokutnika

Primjenom ovog algoritma značajno se olakšava detekcija sudara jer čim se prvi put pronađe os koja razdjeljuje dva objekta, algoritam može stati. Zbog toga on ima dobre performanse u slučaju da je na sceni puno objekata, ali nema puno sudara.

Ovaj algoritam ne može detektirati sudar u slučaju da se objekti pomiču jako velikim brzinama tj. da se objekt pomakne kroz čitavi drugi objekt u samo jednom okviru. Tada pri detekciji se neće detektirati sudar ni u prvom ni u drugom okviru. Kako bi se ovo izbjeglo, potrebno je raditi manje korake pri promjeni pozicije te raditi detekciju za svaki korak (Slika 2.4).



Slika 2.4 Koraci pri rađenju detekcije sudara

Ako želimo dodati novi konveksni primitivni oblik, potrebno je jedino definirati osi koje je potrebno tražiti za detekciju sudara s njime, dok tijela izgrađujemo pomoću skupa već postojećih primitivnih oblika.

U SAT algoritmu poželjno je vraćati informacije o sudaru jer olakšava daljnje računanje u rezoluciji sudara.

Primjer jednostavne implementacije SAT algoritma:

```
function sat(o1, o2){
    let minOverlap = null;
    let smallestAxis;
    let vertexObj;

    let axes = findAxes(o1, o2);
    let proj1, proj2 = 0;
    let firstShapeAxes = getShapeAxes(o1);

    for(let i=0; i<axes.length; i++){
        proj1 = projShapeOntoAxis(axes[i], o1);
        proj2 = projShapeOntoAxis(axes[i], o2);
        let overlap = Math.min(proj1.max, proj2.max) -
Math.max(proj1.min, proj2.min);
        if (overlap < 0){
            return false;
        }
        if((proj1.max > proj2.max && proj1.min < proj2.min)
|| (proj1.max < proj2.max && proj1.min > proj2.min)) {
              let mins = Math.abs(proj1.min - proj2.min);
              let maxs = Math.abs(proj1.max - proj2.max);
              if (mins < maxs){
                  overlap += mins;
              } else {
                  overlap += maxs;
                  axes[i] = axes[i].mult(-1);
              }
          }

        if (overlap < minOverlap || minOverlap === null){
            minOverlap = overlap;
            smallestAxis = axes[i];
            if (i<firstShapeAxes){
                vertexObj = o2;
                if(proj1.max > proj2.max){
                    smallestAxis = axes[i].mult(-1);
                }
            } else {
                vertexObj = o1;
                if(proj1.max < proj2.max){
                    smallestAxis = axes[i].mult(-1);
```

```
                }
            }
        }
    };

    let contactVertex = projShapeOntoAxis(smallestAxis,
vertexObj).collVertex;

    if(vertexObj === o2){
        smallestAxis = smallestAxis.mult(-1);
    }

    return {
        pen: minOverlap,
        axis: smallestAxis,
        vertex: contactVertex
    }
}
```

Programski kod 2.1 – Implementacija SAT algoritma koja vraća podatke o sudaru

2.1.1. SAT algoritam za liniju

SAT algoritam vrlo je lagano izračunati za liniju. Jedina os na koju je potrebno projicirati točke je os koju dobijemo spajajući početnu i krajnju točku linije. Vektor te osi lagano možemo dobiti oduzimanjem vektora pozicija te dvije točke.

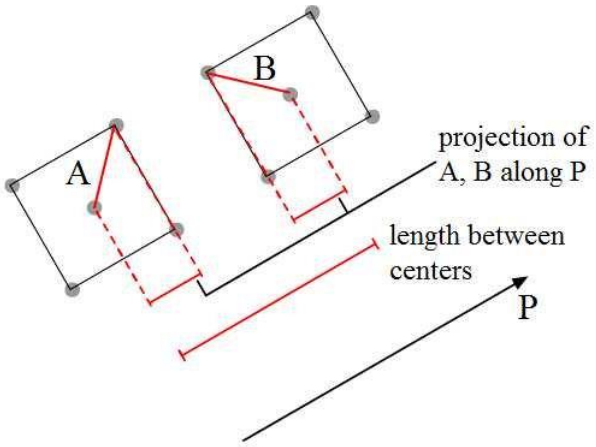
Kod izračuna projekcije na os, potrebno je projicirati samo početnu i krajnju točku na os projekcije te je jedino potrebno pronaći koja je od te dvije točke minimum, a koja maksimum.

2.1.2. SAT algoritam za pravokutnik

Općenito, za četverokut bilo bi potrebno koristiti 4 osi, ali pravokutnik ima dva para osi koje su jednake. Zato je za četverokut potrebno jedino računati dvije osi projekcije.

Te osi moguće je dobiti direktno iz točaka pravokutnika, ali je moguće i izračunati jednu os pa drugu dobiti tako da se izračuna okomita os na nju.

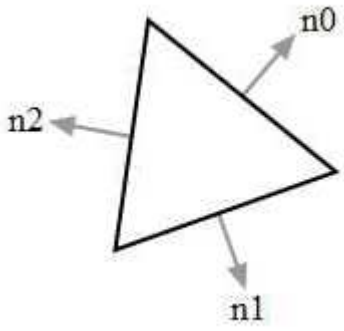
Kod izračuna projekcije na os, potrebno je izračunati projekciju sva 4 vrha pravokutnika te pronaći minimum i maksimum tih točaka na projekciji (Slika 2.5).



Slika 2.5 Izračun SAT algoritma za pravokutnik

2.1.3. SAT algoritam za trokut

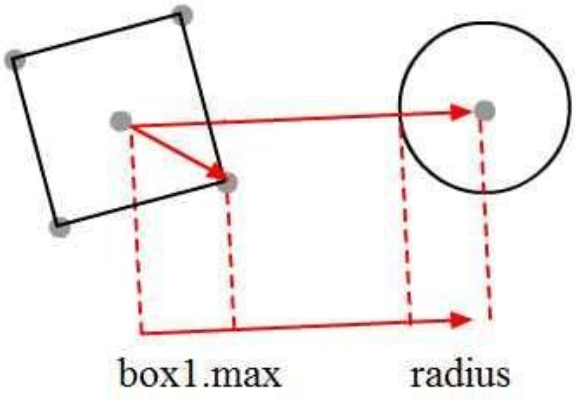
SAT algoritmu potrebne su osi sva tri brida trokuta (Slika 2.6). Bridove možemo dobiti iz spremljenih točaka trokuta. Kao i za pravokutnik, moramo sve točke projicirati na tražene osi pa tražiti minimum i maksimum za svaku os.



Slika 2.6 Izračun osi projekcije za trokut

2.1.4. SAT algoritam za krug

Primjena SAT algoritama za krug je specifična jer nije moguće odrediti os iz objekta samog po sebi. Os projekcije koju je potrebno pronaći mora prolaziti kroz središte kruga te kroz točku drugog objekta koja je najbliža krugu (Slika 2.7). Kada projiciramo krug na os projekcije, možemo projicirati samo središte te dodavanjem radijusa u obje strane iz središta možemo dobiti minimum i maksimum projekcije.



Slika 2.7 SAT algoritam za krug i pravokutnik

2.2. Izračun projekcije točaka na os

Izračun projekcije točaka može se postići pomoću skalarnog umnoška vektora. Dobivanje minimuma i maksimuma skalarnih umnožaka svih točaka dvaju objekta je potrebno kako bi se izvršio SAT algoritam. Skalarni umnožak može se dobiti prema formuli (1):

$$dot = \overrightarrow{point}_x * \overrightarrow{axis}_x + \overrightarrow{point}_y * \overrightarrow{axis}_y \tag{1}$$

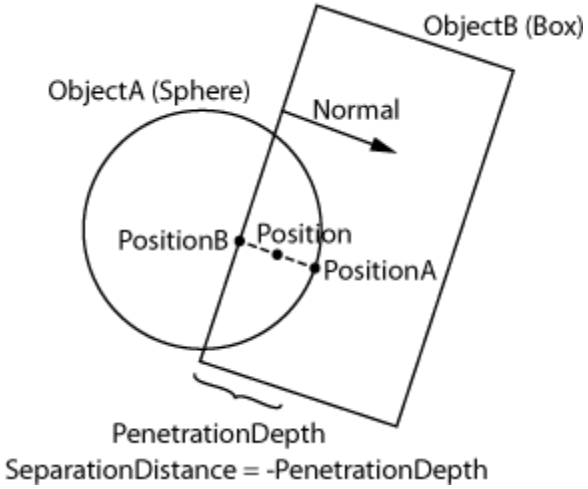
```
function projShapeOntoAxis(axis, obj){
    let min = Vector.dot(axis, obj.vertices[0]);
    let max = min;
    let collVertex = obj.vertices[0];
    for(let i=0; i<obj.vertices.length; i++){
        let p = Vector.dot(axis, obj.vertices[i]);
        if(p<min){
            min = p;
            collVertex = obj.vertices[i];
        }
        if(p>max){
            max = p;
        }
    }
    return {
        min: min,
        max: max,
        collVertex: collVertex
    }
```

```
}
```

Programski kod 2.2 – Projekcija točaka na os s izračunatim minimum i maksimuma

3. Rezolucija sudara

Nakon detekcije sudara, potrebno je odrediti kako će se objekti ponašati u sudaru. Rezolucija sudara mora se odvijati u više koraka jer će se objekti preklapati nakon pomicanja (Slika 3.1).



Slika 3.1 Preklapanje dvaju objekata

Prvi korak rezolucije zato mora biti razdvajanje objekata. Nakon što su objekti razdvojeni, može se računati sudar koristeći fizikalne formule za kruta tijela te koristeći svojstva pojedinih tijela.

3.1. Računanje fizikalnih svojstava

Svakom objektu u sceni potrebna su fizikalna svojstva kako bi se mogao pravilno izračunati sudar s drugim objektima. Međutim, izračun ovih svojstava ovisi o obliku tijela te je potrebno za svaki oblik definirati način kako će se ova svojstva izračunati.

Za izračun mase pretpostavljamo uniformnu razdiobu gustoće po tijelu. Tada za izračun možemo koristiti formulu (2):

$$m = A * \rho \tag{2}$$

Gustoću je moguće tražiti od korisnika, dok površinu je potrebno izračunati na drukčiji način za svaki različiti oblik.

Formula (3) za izračun površine zvijezde:

$$A = r^2\sqrt{3}$$

(3)

Nakon izračuna mase, potrebno je izračunati moment inercije objekta. Moment inercije predstavlja u kojoj mjeri objekt pruža otpor rotaciji. Može se smatrati ekvivalentom mase za rotaciju objekta. Izračun ove vrijednosti nije intuitivan te je težak za određene oblike pa je zato ponekad poželjno koristiti aproksimaciju za ovu vrijednost. Za česte objekte je moguće pronaći formule za izračun ove vrijednosti [6].

Formula (4) za izračun momenta inercije za pravokutnik:

$$I = \frac{1}{12} m(h^2 + w^2)$$

(4)

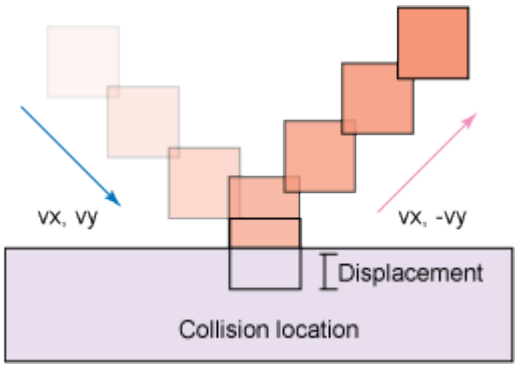
3.2. Razdvajanje objekata

Pri detekciji sudara, dva objekta uvijek će imati nekako preklapanje zbog nedovoljnog uzorkovanja. To preklapanje nam nije prihvatljivo te se objekti moraju razdvojiti.

Razdvajanje objekata mora se odvijati odmicanjem oba objekta preko najkraće moguće osi za odvajanje. Tu je os najlakše odrediti tijekom detekcije sudara SAT algoritmom te možemo spremati informacije potrebne za razdvajanje već tijekom same detekcije.

Osim osi, treba nam i dubina preklapanja da znamo koliko moramo razmaknuti objekte po osi. I dubina se izračunava tijekom SAT algoritma te se može spremiti da daljnje korištenje.

Primjer razdvajanja objekata se može vidjeti na slici (Slika 3.2).



Slika 3.2 Razdvajanje objekata

```
penRes(){
        let penResolution = this.normal.mult(this.pen /
(this.o1.inv_mass + this.o2.inv_mass));
        this.o1.position =
this.o1.position.add(penResolution.mult(this.o1.inv_mass));
        this.o2.position =
this.o2.position.add(penResolution.mult(-this.o2.inv_mass));
}
```

Programski kod 3.1 – Jednostavna implementacija razdvajanja objekata

3.3. Izračun sudara

Izračun sudara svodi se na računanje novog brzine za oba kruta tijela nakon sudara. Nove vrijednosti brzine mogu se pojedinačno računati za brzinu translacije te brzinu rotacije objekta.

3.3.1. Izračun brzina translacije

Pri izračunu brzina, potrebno je spomenuti koeficijent restitucije koji modelira elastičnost objekata. Koeficijent restitucije mora biti između 0 i 1, gdje 0 predstavlja savršeno neelastičan sudar dok 1 predstavlja savršeno elastičan sudar [5]. Koeficijent restitucije računa se prema formuli (5):

$$C_r = -\frac{(\vec{v}_{a+} - \vec{v}_{b+}) * \hat{n}}{(\vec{v}_{a-} - \vec{v}_{b-}) * \hat{n}} \tag{5}$$

Koeficijent restitucije potrebno je odrediti prije sudara te se koristi za daljnje računanje kao omjer relativnih brzina prije i poslije sudara.

Kako bi sudar utjecao više na objekte s manjom masom, a manje na objekte s većom masom, potrebno je izračunati impuls prema formuli (6):

$$j = \frac{(-1 + C_r)[(\vec{v}_{a-} - \vec{v}_{b-}) * \hat{n}]}{(\frac{1}{m_a} + \frac{1}{m_b})} \tag{6}$$

Nakon izračuna udružene mase, moguće je izračunati brzinu objekata nakon sudara. Brzina se može izračunati prema formulama (7) i (8):

$$\vec{v}_{a+} = \vec{v}_{a-} + \frac{j}{m_a}\hat{n} \tag{7}$$

$$\vec{v}_{b+} = \vec{v}_{b-} - \frac{j}{m_b}\hat{n} \tag{8}$$

3.3.2. Izračun brzina rotacije

Ukoliko računamo i rotaciju objekata, vektor brzine će nam izgledati kao (9) i (10):

$$\overline{v_a} = \vec{v}_{a-} + \times \vec{r}_a \tag{9}$$

$$\overline{v_b} = \vec{v}_{b-} + \vec{\omega}_{b-} \times \vec{r}_b \tag{10}$$

gdje $\vec{\omega}_-$ označava brzinu rotacije prije sudara, a $\vec{r}$ označava vektor od centra rotacije do točke sudara.

Formula za koeficijent restitucije tada izgleda kao (11):

$$C_r = -\frac{(\overline{v_{a+}} - \overline{v_{b+}}) * \hat{n}}{(\overline{v_{a-}} - \overline{v_{b-}}) * \hat{n}}$$

(11)

Formula za impuls koja uključuje i inerciju objekata (12):

$$j = \frac{(-1 + C_r)[(\overline{v_{a-}} - \overline{v_{b-}}) * \hat{n}]}{\left(\frac{1}{m_a} + \frac{1}{m_b}\right) + [(\vec{r}_a \times \hat{n})^T {I_a}^{-1} (\vec{r}_a \times \hat{n}) + (\vec{r}_b \times \hat{n})^T {I_b}^{-1} (\vec{r}_b \times \hat{n})]}$$

(12)

Sada jedino ostaje izračunati nove vrijednosti brzine rotacije. Brzine rotacije se mogu izračunati prema formulama (13) i (14):

$$\vec{\omega}_{a+} = \vec{\omega}_{a-} + {I_a}^{-1}(\vec{r}_a \times j\hat{n})$$

(13)

$$\vec{\omega}_{b+} = \vec{\omega}_{b-} + {I_b}^{-1}(\vec{r}_b \times j\hat{n})$$

(14)

Primjer moguće implementacije izračuna sudara:

```
collRes(){
        let collArm1 =
this.cp.subtr(this.o1.components[0].position);
        let rotVel1 = new Vector(-this.o1.angVelocity *
collArm1.y, this.o1.angVelocity * collArm1.x);
        let closVel1 = this.o1.velocity.add(rotVel1);
        let collArm2 =
this.cp.subtr(this.o2.components[0].position);
        let rotVel2= new Vector(-this.o2.angVelocity *
collArm2.y, this.o2.angVelocity * collArm2.x);
        let closVel2 = this.o2.velocity.add(rotVel2);

        let impAug1 = Vector.cross(collArm1, this.normal);
        impAug1 = impAug1 * this.o1.inv_inertia * impAug1;
        let impAug2 = Vector.cross(collArm2, this.normal);
        impAug2 = impAug2 * this.o2.inv_inertia * impAug2;
```

```
                let relVel = closVel1.subtr(closVel2);
                let sepVel = Vector.dot(relVel, this.normal);
                let new_sepVel = -sepVel *
            Math.min(this.o1.elasticity, this.o2.elasticity);
                let vsep_diff = new_sepVel - sepVel;

                let impulse = vsep_diff / (this.o1.inv_mass +
            this.o2.inv_mass + impAug1 + impAug2);
                let impulseVec = this.normal.mult(impulse);
            this.o1.velocity =
this.o1.velocity.add(impulseVec.mult(this.o1.inv_mass));
                this.o2.velocity =
            this.o2.velocity.add(impulseVec.mult(-this.o2.inv_mass));

                this.o1.angVelocity += this.o1.inv_inertia *
            Vector.cross(collArm1, impulseVec);
                this.o2.angVelocity -= this.o2.inv_inertia *
            Vector.cross(collArm2, impulseVec);
            }
```

Programski kod 3.2 – Rezolucija sudara

4. Interakcija s korisnikom

Fizikalni pogon ponekad zahtijeva interakciju s korisnikom. Ta interakcija može se odvijati na razne načine, ali najčešći je odabir interakcije putem miša i tipkovnice.

Javascript ima mogućnosti dodavanja *callback* funkcija za određene događaje. Za potrebe ovog rada, trebaju se dodati funkcije za interakciju s platnom. Funkcije za klik miša trebaju koristiti koordinate klika na platno koje se moraju izračunati preko apsolutne pozicije miša te pozicije samog platna na ekranu. Funkcije za interakciju s tipkovnicom moraju provjeravati koja je tipka pritisnuta te pokrenuti potrebnu akciju.

4.1. Odabir objekta

Odabir objekta iz koordinata miša na platnu mora pronaći objekt koji je na tom mjestu iscrtan na platnu. Postoje mnogi načini kako provjeriti koji objekt treba biti odabran s raznim performansama te raznom kompleksnosti. Jedan način koji je vrlo lagan za implementaciju je stavljanje točke na mjesto klika te provjera sudara točke s drugim objektima na sceni. Ako se točka sudara s objektom, onda taj objekt treba odabrati. Nakon toga, točka se obriše sa scene (točka se obriše prije iscrtavanja scene).

```
            let clickBall = new Ball(mouseX, mouseY, 5, 0);
            let found = false;
            BODIES.forEach((b, index) => {
                if(collide(b, BODIES[BODIES.indexOf(clickBall)])
!= false && found === false) {
                    BODIES[index].player = true;
                    found = true;
            }
            else {
                BODIES[index].player = false;
            }
            });
            clickBall.remove();
```

Programski kod 4.1 – Odabir objekta

4.2. Dodavanje objekata

Dodavanje objekta odvija se na sličan način kao i odabir objekta. Za razliku od odabira, od korisnika se treba tražiti više podataka kako bi se znali izračunati svi potrebni podatci za iscrtavanje i ponašanje tog objekta (npr. boja, gustoća).

Prilikom dodavanja objekata poželjno je provjeriti je li taj objekt u sudaru s drugim objektima kada bi se iscrtao. U suprotnom bi moglo doći do slučaja gdje nema mjesta na sceni, ali se svejedno dodaju novi objekti. Ako se objekt u sudaru, potrebno je ili javiti korisniku da ne može na tom mjestu dodati objekt ili ga pomaknuti tako da više nije u sudaru.

```
        let clickBall = new Ball(mouseX, mouseY, 5, 0);
        let found = false;
        let object;
        BODIES.forEach((b, index) => {
            if(collide(b, BODIES[BODIES.indexOf(clickBall)])
!= false && found === false) {
                object = b;
                found = true;
        }
        });
        clickBall.remove();
        if(found)
            object.remove();
```

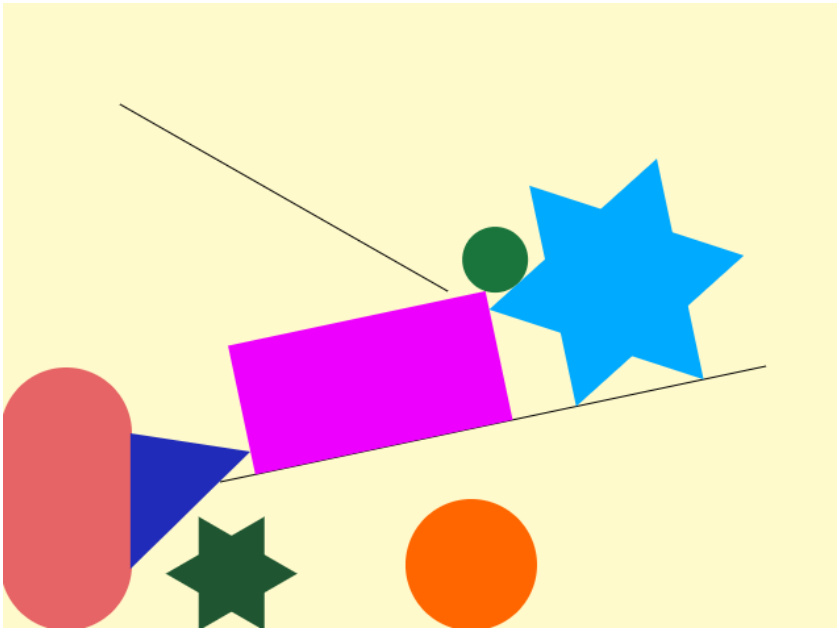
Programski kod 4.2 – Dodavanje kugle u scenu

5. Rezultati i moguća poboljšanja

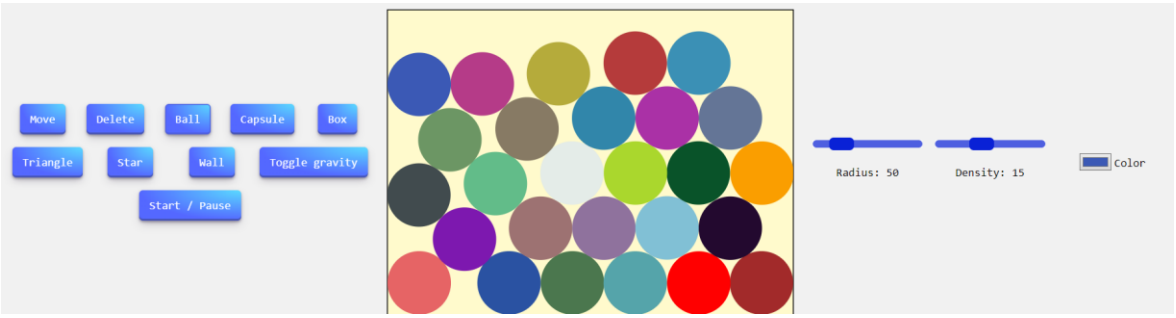
U trenutnoj implementaciji fizikalni pogon ima sljedeće značajke:

- 6 različitih oblika (kugla, kapsula, pravokutnik, trokut, zvijezda, zid)
- Mogućnost pomaka objekata, dodavanje novih objekata te brisanje
- Uključenje / isključenje gravitacije te same simulacije
- Vjeran prikaz sudara objekata na sceni
- 60 okvira po sekundi
- Lagano proširenje pogona te dodavanje novih vrsta oblika

Izgled implementacije je vidljiv na slikama Slika 5.1 i Slika 5.2



Slika 5.1 Platno fizikalnog pogona



Slika 5.2 Cijeli zaslon fizikalnog pogona

5.1. Moguća poboljšanja

Lista mogućih poboljšanja za fizikalni pogon:

- Dodavanje više vrsta različitih oblika
- Mijenjanje svojstva postojećih oblika
- Više interakcije s korisnikom
- Korištenje algoritama s boljim performansama
- Implementacija algoritama za detekciju sudara koji dobro funkcioniraju i za konkavna tijela
- Dodavanje više fizikalnih simulacija poput njihala, opruga, mekih tijela te mnogih drugih

Zaključak

Fizikalni pogon simulacije krutih tijela izvršava se sljedećim redoslijedom:

1. Pozicijska logika
2. Detekcija sudara
3. Rezolucija sudara

Pozicijska logika i rezolucija sudara temelje se na formulama mehanike krutih tijela, dok se detekcija sudara temelji na algoritmima iz računalne znanosti.

SAT algoritam vrlo je popularan algoritam za ovakve fizikalne pogone te može detektirati sudare za sve konveksne poligone te sve poligone koji se mogu rastaviti na skup konveksnih poligona. Idealan je za slučajeve gdje u sceni ima puno objekata, ali nema puno sudara. Za izračun su potrebni svi vrhovi konveksnog poligona te osi projekcije potrebne za taj poligon. Kada se svi vrhovi projiciraju na os, računa se maksimum i minimum za tu os. Izračun projekcije može se napraviti koristeći skalarni produkt vektora osi te vektora točke.

Pri rezoluciji sudara, potrebno je prvo razdvojiti objekte jedan od drugoga po najkraćoj osi te je tek onda moguće računati sudar. Za računanje sudara, prvo moramo odrediti fizikalna svojstva svih tijela na sceni. Tada je moguće računati novi vektor brzine. Vektori translacije objekta te rotacije objekta nakon sudara računaju se pojedinačno. Omjer relativnih brzina prije i nakon sudara određen je koeficijentom restitucije. Tijela s većom masom imati će manju promjenu vektora brzine, dok tijela s manjom masom će imati veću promjenu vektora brzine.

Nakon svih izračuna, objekte na sceni je moguće iscrtavati.

Interakcija a korisnikom u fizikalnom pogonu često se odvija putem miša i tipkovnice. Za interakciju s mišem, potrebno je izračunavati poziciju miša unutar platna koristeći apsolutnu poziciju miša te trenutnu poziciju platna unutar zaslona. Za interakciju s tipkovnicom, potrebno je provjeriti koja je tipka na tipkovnici pritisnuta.

Kod odabira objekta interakcije, potrebno je provjeriti s kojim je objektom sudara trenutna pozicija miša te njega odabrati. Prilikom dodavanja objekata, potrebno je paziti na slučaj ako bi taj objekt se sudario s nekim drugim objektom pri dodavanju.

Literatura

[1] *Build a simple 2D physics engine for JavaScript games,* IBM developer. Poveznica: https://developer.ibm.com/tutorials/wa-build2dphysicsengine/

[2] *How to Create a Custom 2D Physics Engine: The Basics and Impulse Resolution*, Tuts+. Poveznica: https://gamedevelopment.tutsplus.com/tutorials/how-to-create-a-custom-2d-physics-engine-the-basics-and-impulse-resolution--gamedev-6331

[3] Tomislav, O. *Detekcija sudara čvrstih tijela*. Diplomski rad. Sveučilište u Zagrebu Fakultet elektrotehnike i računarstva, 2010.

[4] *Separating Axis Theorem*, Programmer Art. Poveznica: http://programmerart.weebly.com/separating-axis-theorem.html

[5] Saša Ilijić, *Mehanika – pojmovi, načela i odabrani primjeri*. 1. izdanje. Zagreb: Vlastita naklada autora, 2020., str. 69-85.

[6] *List of moments of inertia*, Wikipedia. Poveznica: https://en.wikipedia.org/wiki/List_of_moments_of_inertia

[7] *matter.js*, brm.io. Poveznica: https://brm.io/matter-js/

Sažetak

2D fizikalni pogon

Sažetak

U ovom radu, prikazane su faze izrade fizikalnog pogona za simulaciju krutih tijela te fizikalna podloga potrebna za njegovu izradu. Implementacija je pisana u jeziku Javascript. Pogon radi u stvarnom vremenu te pretpostavlja da je između dva okvira uvijek prošlo jednako vrijeme (60 okvira u sekundi). Opisani su algoritmi detekcije sudara konveksnih objekata te formule za rezoluciju sudara. Algoritam korišten za detekciju sudara je SAT algoritam, a fizikalne formule dobivene su iz mehanike krutih tijela. Fizikalni pogon implementira interakciju s korisnikom te sadrži kratki opis potrebnih funkcija za interakciju.

Ključne riječi: 2D, fizikalni pogon, pogon u stvarnom vremenu, Javascript, statički pogon, kruta tijela, sudari, detekcija sudara, SAT algoritam, mehanika, interakcija s korisnikom

Summary

2D physics engine

Summary

In this thesis, steps of making a physics engine for rigid bodies and the physics behind it are presented. The implementation was written in Javascript. The engine works in real time and assumes that the same time has passed between two frames (60 frames a second). Collision detection algorithms for convex polygons and collision resolution are described. Separation axis theorem algorithm was used for collision detection and the physics formulas used for collision resolution were obtained from rigid body mechanics. The engine implements basic user interaction and contains a short description of functions needed to obtain it.

Keywords: 2D, physics engine, real-time engine, Javascript, static engine, rigid bodies, collisions, collision detection, SAT algorithm, mechanics, user interaction

Privitak

Programski kod se može pronaći na poveznici: https://github.com/Samcina/2d-physics-engine.