FUTUREMATH ERASMUS+ PROJECT

Co-funded by the Erasmus+ Programme of the European Union

UNIVERZITET METROPOLITAN BEOGRAD

CS232 - PROGRAMIRANJE 2D IGARA

Fizika Igara - diferencijalne jednačine kretanja

Lekcija 16

PRIRUČNIK ZA STUDENTE

CS232 - PROGRAMIRANJE 2D IGARA

Lekcija 16

FIZIKA IGARA - DIFERENCIJALNE JEDNAČINE KRETANJA

UVOD

Računarska igra ima potrebu da dogadjaju u igri liče na stvarne. To nije moguće ostvariti bez primene fizike u igrama

Računarska igra, kao i animacija ili film, ima potrebu da događaji u igri liče na one iz realnosti. Postoje dva pokazatelja kvaliteta sličnosti virtuelnog (onog što se događa u na ekranu tokom igre) i realnog sveta, a to su realizam i vernost. Nivo realizma i vernosti u video igri bi trebalo da bude što veći. Pod vernošću se podrazumeva kvalitet predstava brzina, ubrzanja i položaja objekata u vremenu. Ovim se bavi oblast koja se naziva fizika igara.

U video igri obično pokušavamo da u petlji prikazujemo kretanja objekata, i to fejm po frejm. Postavlja se pitanje kako reći računaru gde da stavi objekte? Pa, zakoni fizike upravo kažu kako se objekti kreću.

Kod igara kreiranih 70tih ili 80tih godina fizika nije nalazila veliku primenu. Međutim, početkom 90tih, u takozvanoj 3D eri, fizičko modeliranje je postalo veoma bitno. Jednostavno se nije moglo zamisliti da se objekti u video igi ne pomeraju na realističan način — morali su da se kreću približno jednako kao u realnom svetu.

Svako telo koje se kreće pravolinijski i konstantnom brzinom teži da zadrži to stanje kretanja. Da bi telo promenilo to stanje kretanja (tj. ubrzalo ili usporilo), potrebno je da se deluje na njega iz okoline, a ta akcija se naziva sila. Stoga ćemo se baviti Njutnovim zakonima koji opisuju kretanje tačaka i tela pod dejstvom sila, i implementacijom u programskom jeziku.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Poglavlje 1

Ravnomerno kretanje

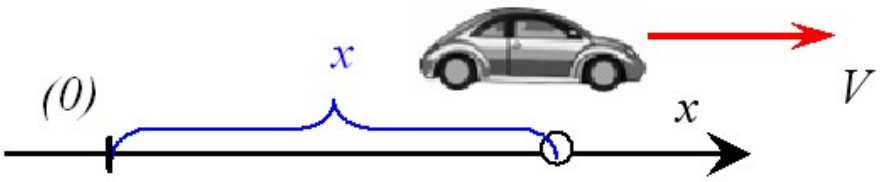
JEDNODIMENZIONALNO KRETANJE

Brzina je pređeni put u jedinici vremena. Srednja brzina je količnik pređenog puta i proteklog vremena

Neka se objekat kreće duž prave (ili krive) na kojoj smo izabrali tačku x, od koje ćemo meriti pređeno rastojanje, i koordinatni sistem *(O,x)*.

Neka se u početnom trenutku *t=0* telo nalazi u početnom položaju *x = 0*, i neka se u proizvoljnom trenutku *t* telo nalazi na rastojanju *x* od početnog položaja. Podsetimo se da je:

Brzina = pređeni put u jedinici vremena.



Slika 1.1 Brzina je pređeni put u jedinici vremena [2]

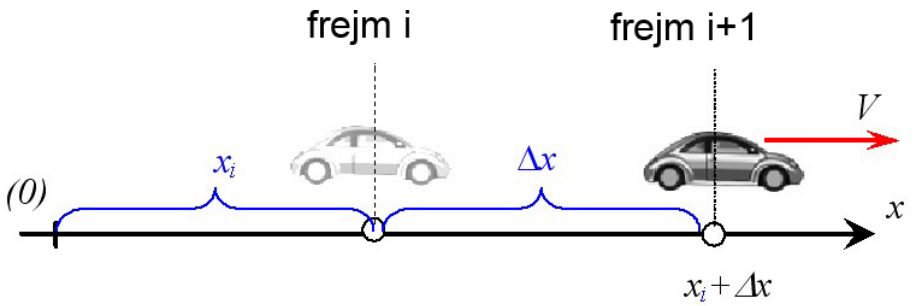
Neka je od i-tog do (i+1)-og frejma proteklo vreme $\triangle t$ (tj. jedan frejm traje $\triangle t$), i neka je telo za to vreme prešlo rastojanje $\triangle x$.

Srednju brzinu kretanja tokom intervala ćemo sračunati kao pređeno rastojanje u jedinici vremena:

$$V_{sr} = \frac{\triangle x}{\triangle t}$$

Ako znamo srednju brzinu tokom (i+1)-og frejma, poziciju (position) za prikazivanje na ekranu možemo sračunati kao:

$$x_{i+1} = x_i + \triangle x = x_i + V_{sr} \triangle t$$



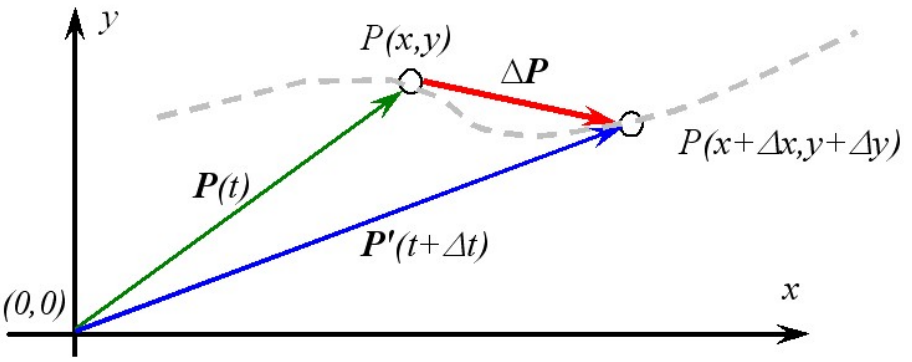
Slika 1.2 Srednja brzina je pređeni put u proteklom vremenu [2]

DVODIMENZIONALNO KRETANJE

Brzina kretanja je odnos promene položaja tela i proteklog vremena. Trenutna brzina je izvod vektora pozicije po vremenu

Neka se telo kreće i neka u trenutku *t* zauzima položaj ***P****(t)*. Neka se nakon vremena Δ*t* telo pomerilo za vektor Δ***P***, pri čemu su se njegove koordinate promenile za Δ*x* iΔ*y*, tako da zauzima položaj ***P'***.

$$\mathbf{P}' = \begin{bmatrix} \mathbf{x} + \triangle \mathbf{x} \\ y + \triangle y \end{bmatrix}$$



Slika 1.3 Promena položaja tela u 2D [izvor: autor]

Srednju brzinu kretanja tačke definišemo kao promenu pozicije u jedinici vremena. Kako je promena pozicije vektorska veličina, to će i brzina biti vektor:

$\boldsymbol{V}_{sr} = \Delta\mathbf{P}/\Delta t$

Kako se telo kreće duž osa x i y, to se mogu definisati srednje brzine:

$V_{xsr} = \Delta x / \Delta t$

$V_{ysr} = \Delta y / \Delta t$

koje predstavljaju skalarne veličine. Naravno, pritom važi:

$$\mathbf{V_{sr}} = \begin{bmatrix} V_{xsr} \\ V_{ysr} \end{bmatrix}$$

Trenutna brzina (ili vrednost srednje brzine kad $\Delta t \to 0$) je izvod vektora pozicije ***P*** po vremenu:

$\boldsymbol{V} = d\boldsymbol{P} / dt$

Koordinate vektora brzine ***V*** su V_x i V_y i predstavljaju brzine pomeranja tela u pravcu osa *x* i *y.* Imamo da je:

$V_x = d\boldsymbol{x} / dt$, $V_y = d\boldsymbol{y} / dt$

Napomena

Vektor brzine je uvek tangentana na putanju kojom se telo kreće .

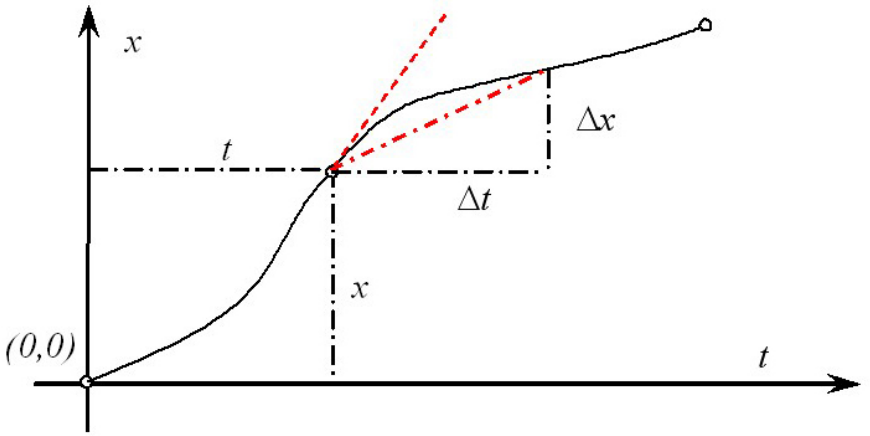
TRENUTNA BRZINA

Trenutna brzina je predjeni put u beskonačno kratkom vremenu dt

Ako želimo tačnu brzinu u trenutku *t* (takozvanu trenutnu brzinu), tada treba smanjivati Δt.

Kad $\Delta t \to 0$ (tj. Δt teži nuli) dobija se da je trenutna brzina u trenutku *t:*

$V = dx/dt$



Slika 1.4 Trenutna brzina je predjeni put u beskonačno kratkom vremenu dt [2]

tj.

Trenutna brzina je jednaka izvodu pređenog rastojanja po vremenu.

Ako nacrtamo grafik pozicije *x* u funkciji pređenog vremena *t*, tada je trenutna brzina jednaka nagibu tangente na dobijenu krivu u tački *(t,x)* , dok je srednja brzina jednaka nagibu sečice kroz datu tačku (Slika-3).

Važi takođe da je *pozicija* ***integral trenutne brzine po vremenu*** .

PRIMENA BRZINE U IGRAMA

Brzina i trajanje frejma određuju promenu pozicije (update) tokom frejma

Za igre je puno interesantnija sledeća intepretacija. Izaberimo interval posmatranja Δt (odnosno trajanje frejma) dovoljno mali tako da se brzina veoma V malo menja u toku Δt. Neka je V_i brzina u i-tom frejmu, pređeno rastojanje u i-tom frejmu je jednako $V_i \Delta t$. Da bi se dobili ukupni put i vreme nakon n frejmova, treba sumirati priraštaje u svakom frejmu:

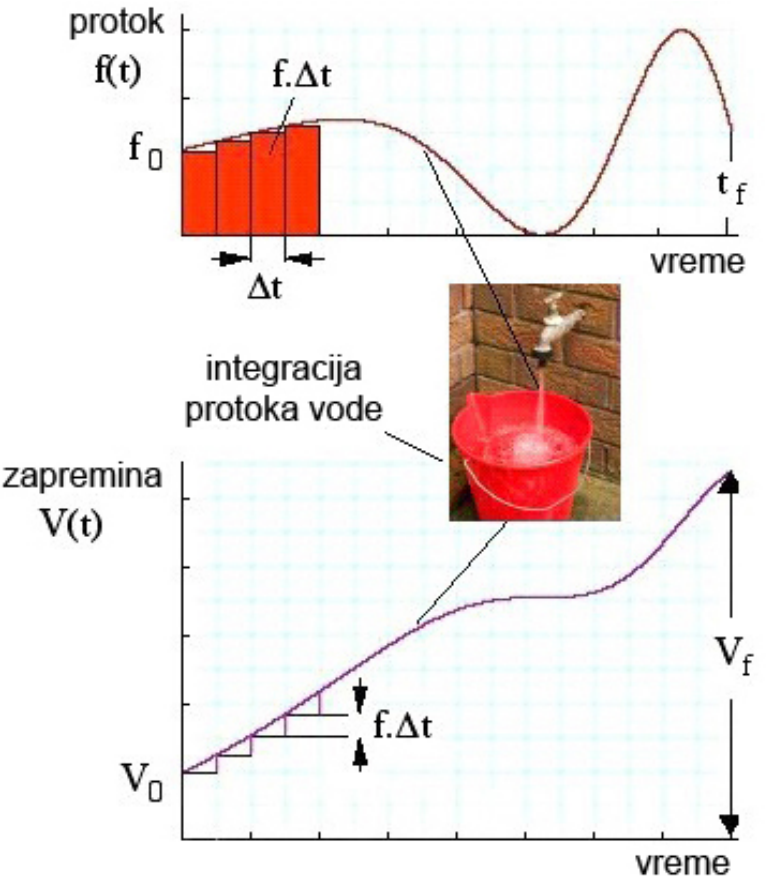
$x = \Sigma_i V_i \Delta t$, $t = \Sigma_i \Delta t$, $i = 1,2, \ldots, n$

U igri ili simulaciji, integraljenje (sumiranje) se odvija iterativno tako da se u svakom frejmu sračunava tekuća pozicija:

```
// Inicijalizacija
float x = 0, t = 0,
V = V_initial, dt = 0.033;
while(!game_over)
{
    t = t + dt ;
    V = get_velocity ( t) ;
    x = x + V*dt ;
}
```

Primer

Na Slici-4 je prikazano punjenje kofe vodom koje se ponaša po istom matematičkom modelu. Dotok vode (flow) je f litara u sekundi. Otuda, za vreme Δt u kofu uđe $f\,\Delta t$ litara. Sumiranjem se dobije količina vode nakon željenog broja frejmova.



Slika 1.5 Analogija integrala sa punjenjem suda

Poglavlje 2

Ubrzano kretanje

UBRZANJE

Promena brzine u jedinici vremena se naziva ubrzanje

Ako je brzina nekog kretanja konstantna tokom intervala posmatranja od 0 do t,

$V = const$

tada je pozicija tela u trenutku t jednaka

$x = V\,t$

ako se u početnom trenutku $t = 0$ telo nalazilo na poziciji $x = 0$.

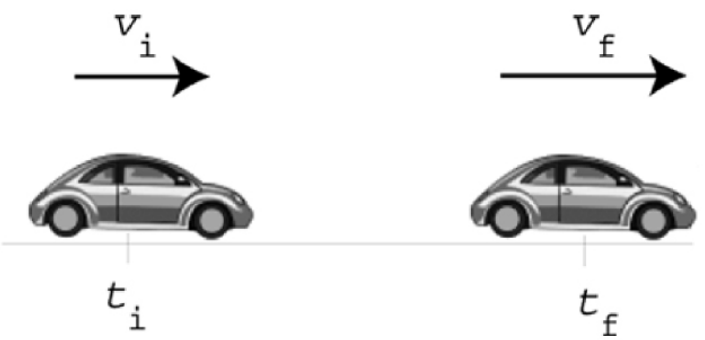
Ako je pri $t = 0$ telo bilo u poziciji $x = x_0$ tada je

$x = x_0 + V\,t$

Napomena

Upravo ova relacija je iskorišćena za osvežavanje pozicije tokom frejma, jer se može smatrati da je trajanje frejma dovoljno kratko da se brzina neće promeniti tokom frejma.

U opštem slučaju se brzina menja od frejma do frejma, a ***promena brzine u jedinici vremena se naziva ubrzanje*** .



Slika 2.1 Ubrzanje automobila. Promena brzine između dva vremenska trenutka [2]

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

SREDNJE UBRZANJE

Srednje ubrzanje je količnik promene brzine i proteklog vremena

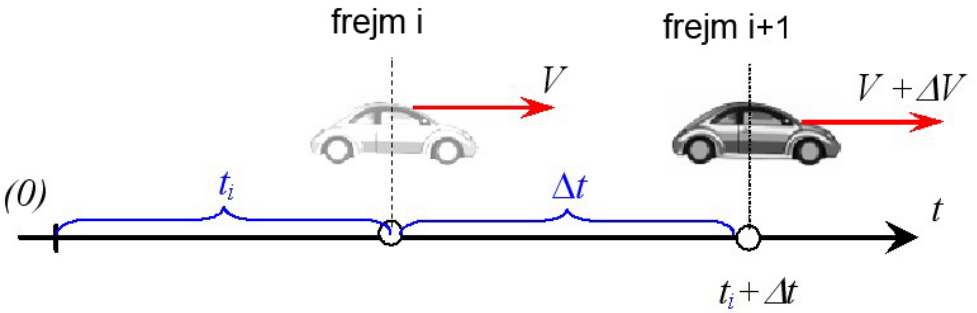
Neka je u frejmu i (odnosno trenutku t_i) brzina tela jednaka V, a neka je u frejmu (i+1) tj. trenutku $t_i + \triangle t$ brzina tela jednaka $V + \triangle V$, Slika-2.

Srednje ubrzanje tokom intervala $\triangle t$ je jednako:

$$a_{sr} = \frac{\triangle V}{\triangle t}$$

Ako znamo srednje ubrzanje tokom (i+1)-og frejma, brzinu za prikazivanje na ekranu možemo sračunati kao:

$$V_{i+1} = V_i + \triangle V = V_i + a_{sr} \triangle t$$



Slika 2.2 Ubrzanje je promena brzine u jedinici vremena [2]

Poziciju tokom (i+1)-og frejma možemo ažurirati stavljajući da je srednja brzina jednaka:

$$V_{sr} = (V_i + V_{i+1})/2 = V_i + a_{sr} \triangle t/2$$

tako da je:

$$x_{i+1} = x_i + \triangle x = x_i + V_{sr} \triangle t$$

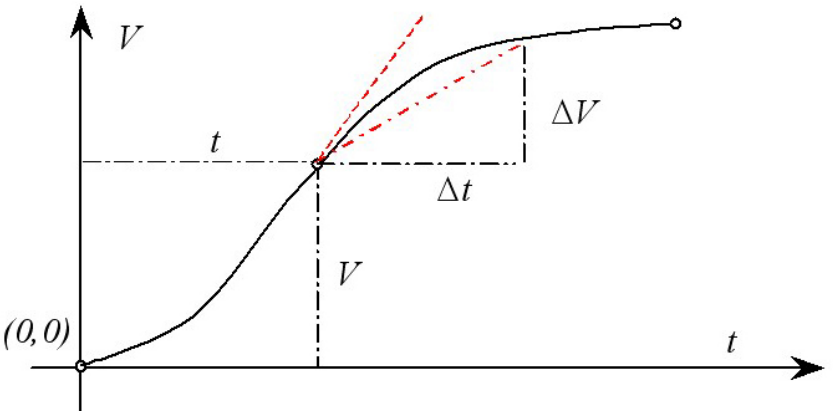
TRENUTNO UBRZANJE

Trenutno ubrzanje je jednako izvodu brzine po vremenu

Ako želimo tačno ubrzanje u trenutku *t* (takozvano trenutno ubrzanje), tada treba smanjivati Δ*t*.

Kad Δ*t* → 0 (tj. kada teži nuli Δt)dobija se da je trenutno ubrzanje u trenutku *t*

a = dV / dt



Slika 2.3 Trenutno ubrzanje je promena brzine u beskonačno kratkom vremenu dt [2]

tj.

Trenutno ubrzanje je jednako izvodu brzine po vremenu.

Obrnuto,

Brzina je integral trenutnog ubrzanja po vremenu.

Ako nacrtamo grafik brzine *V* u funkciji vremena *t*, trenutno ubrzanje je jednako nagibu tangente na dobijenu krivu u tački *(t,V)* , dok je srednje ubrzanje jednaka nagibu sečice kroz datu tačku (Slika-3).

Ubrzanje **a** je vektorska veličina, njegove koordinate su a_x i a_y i predstavljaju ubrzanja tela u pravcu osa *x* i *y.* Imamo da je:

$$a_x = \frac{\mathrm{d}V_x}{\mathrm{d}t}, a_y = \frac{\mathrm{d}V_y}{\mathrm{d}t}$$

Poglavlje 3

Programiranje kretanja u 2D

RAVNOMERNO KRETANJE

Koraci algoritma za programiranje kretanja su: 1. Dodaj brzinu na postojeću lokaciju 2. Iscrtaj objekat na toj lokaciji

Ono sto nas ovde zanima je šta to znači da isprogramiramo kretanje korišćenjem vektora. Jedan objekat na ekranu ima poziciju (mesto u kome se nalazi u datom trenutku) kao i brzinu (instrukcije za način kako će se pomeriti iz jednog momenta u neki drugi). Predjeni put je brzina tela kojr se kreće pomnožena sa proteklim vremenom:

```
location.add(velocity * dT);
```

pri čemu je dT proteklo vreme. Zatim možemo da iscrtamo objekat na toj lokaciji:

```
ellipse(location.x,location.y,16,16);
```

Ovo je Motion 101.
1. Dodaj promenu položaja (v*dT) na trenutnu lokaciju
2. Iscrtaj objekat na toj lokaciji

U primeru lopte koja poskakuje, imamo dve glavne funkcije koje služe za podešavanje i iscrtavanje:*setup()* i *draw()*. Ono što želimo da uradimo u nastavku jeste da celokupnu logiku kretanja učaurimo unutar neke klase. Na ovaj način, kreiramo temelj za programiranje kretanja objekata.

U ovom slučaju, kreiraćemo klasu *Mover* pomoću koje opisujemo stvari koje mogu da se kreću po ekranu. U tu svrhu moramo da razmotrimo sledeća pitanja: koje podatke i funkcionalnost treba da ima klasa *Mover*?

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

KLASA MOVER

Kreiramo klasu Mover pomoću koje opusujemo stvari koje mogu da se kreću po ekranu

Objekat tipa *Mover* ima dva podatka: *location* i *velocity*, koji su oba tipa *PVector*.

```
class Mover
{
public:
    PVector location;
    PVector velocity;
```

Funkcionalnost klase *Mover* treba da bude što prostija. *Mover* treba da se pomera i da bude vidljiv. Ove potrebe ćemo implementirati u okviru funkcija *update()* i *display()*. Celokupnu logiku kretanja ćemo smestiti u funkciju *update()* a iscrtavanje objekta ćemo da smestimo u funkciju *display()*.

```
void update() {
        location.add(velocity * dt); // The Mover moves.
    }

    void display() {
        stroke(0);
        fill(175);
        ellipse(location.x,location.y,16,16); // The Mover is displayed.
    }
}
```

Zaboravili smo jednu veoma bitnu stvar, a to je konstruktor objekta. Konstruktor se poziva kada kreiramo novi objekat naše klase, npr sa:

```
Mover m();
```

U našem slučaju, izaberimo nasumičnu brzinu i lokaciju pomoću kojih ćemo inicijalizovati podatke našeg *Mover* objekta.

```
Mover() {
        location = PVector(random(width),random(height));
        velocity = PVector(random(-2,2),random(-2,2));
    }
```

Privedimo kraju našu klasu *Mover* ugrađivanjem funcija koje proveravaju da li je objekat dostigao ivice prozora aplikacije u kome se vrši njegovo iscrtavanje. Prosta funkcija koja vrši proveru bi imala sledeći oblik.

```
void checkEdges()
    {
        // When it reaches one edge, set location to the other.
        if (location.x > width) {
            location.x = 0;
        } else if (location.x < 0) {
            location.x = width;
        }
        if (location.y > height) {
            location.y = 0.0;
        } else if (location.y < 0) {
            location.y = height;
```

```
        }
    }
```

PRIMER: RAVNOMERNO KRETANJE

Pogledajmo šta je neophodno da uradimo u našem programu da bi simulirali kretanje

Sada kada smo završili klasu *Mover*, možemo da pogledamo šta je neophodno da odradimo u našem programu. Prvo deklarišemo objekat klase *Mover*:

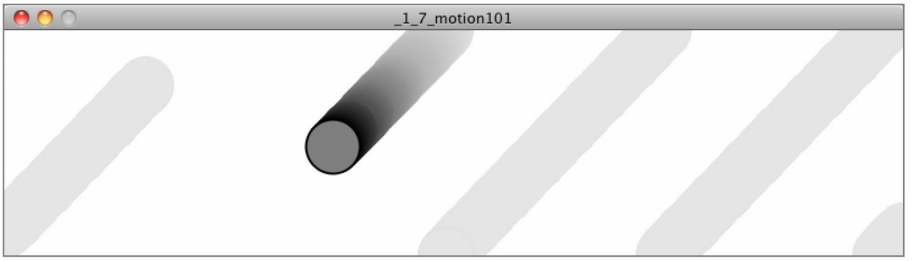
```
Mover mover;
```

Zatim ga inicijalizujemo u funkciji *setup()*:

```
mover = Mover();
```

i pozovemo odgovarajuće funkcije u okviru funkcije *draw()*:

```
mover.update();
mover.checkEdges();
mover.display();
```



Slika 3.1 Primer kretanja u 2D [6]

U nastavku je kompletan kod primera sa Slike-1:

```
// main.cpp:
// L11 - Primer 1: Kretanje 101 (brzina)

Mover mover;   // Declare Mover object.

void setup()
{
    size(200,200);
    setTimer(30);

    mover = Mover();  // //  Create Mover object.
}
```

```
void draw()
{
    background(255);

    mover.update(); // Call functions on Mover object.
    mover.checkEdges();
    mover.display();
}

// Mover.h:
class Mover
{
public:
    PVector location;
    PVector velocity;

    Mover() {
        location = PVector(random(width),random(height));
        velocity = PVector(random(-2,2),random(-2,2));
    }

    void update() {
        location.add(velocity * dt); //The Mover moves.
    }

    void display() {
        stroke(0);
        fill(175);
        ellipse(location.x,location.y,16,16); // The Mover is displayed.*/
    }

    void checkEdges()
    {
        // When it reaches one edge, set location to the other.
        if (location.x > width) {
            location.x = 0;
        } else if (location.x < 0) {
            location.x = width;
        }
        if (location.y > height) {
            location.y = 0.0;
        } else if (location.y < 0) {
            location.y = height;
        }
    }
};
```

UBRZANO KRETANJE

Algoritmi za određivanje ubrzanja koja ćemo obraditi su: konstantno ubrzanje, potpuno slučajno ubrzanje i ubrzanje koje prati miša

Do sada smo se upoznali sa dve stvari: (1) šta je *PVector* i (2) kako koristimo tip *PVector* unutar objekta da bi pratili njegu lokaciju u njegovo kretanje. Međutim, moramo da napravimo još neke bitne korake. Nakon svega, primećujemo da naš primer izgleda dosadno — krug nikada ne ubrzava, usporava, niti se okreće. Za interesantnije kretanje, kretanje koje je bliže realnom svetu koji se dešava oko nas, neophodno je da dodamo još jedan *PVector* zu našu klasu —*acceleration*. U kodu, to bi izgledalo ovako:

```
velocity.add(acceleration * dt);
location.add(velocity * dt);
```

Kao jednu korisnu vežbu, za sve što radimo od sada na dalje, kreirajmo neka pravila kojih ćemo se pridržavati. Pokušajmo da napravimo svaki primer u nastavku bez direktnog dodira sa vrednošću brzine i lokacije (osim kad ih inicijalizujemo). Drugim rečima, naš cilj u toku programiranja kretanja je da osmislimo algoritam za određivanje ubrzanja. Neki od načina da to uradimo su sledeći:
Algoritmi za određivanje ubrzanja!
1. Konstantno ubrzanje
2. Potpuno slučajno ubrzanje
3. Ubrzanje koje prati miša

```
void update() {
        // Our motion algorithm is now two lines of  code!
        velocity.add(acceleration * dt);
        location.add(velocity * dt);
    }
```

Poglavlje 4

Kretanje pod dejstvom sila

SILA I DRUGI NJUTNOV ZAKON

Da bi telo promenilo to stanje kretanja (tj. ubrzalo ili usporilo), potrebno je da se deluje na njega iz okoline, a ta akcija se naziva sila. Jedinica za silu je Njutn

Svako telo koje se kreće pravolinijski i konstantnom brzinom teži da zadrži to stanje kretanja.

Ovo je poznato kao *zakon inercije*. Da bi telo promenilo to stanje kretanja (tj. ubrzalo ili usporilo), potrebno je da se deluje na njega iz okoline, a ta akcija se naziva *sila*.

Poznati *II Njutnov zakon* kaže da je:

$F = m\,a$

gde je ***F*** sila, *m* masa tela, a *a* ubrzanje.

Jedinica za silu je Njutn (skraćeno [N]).

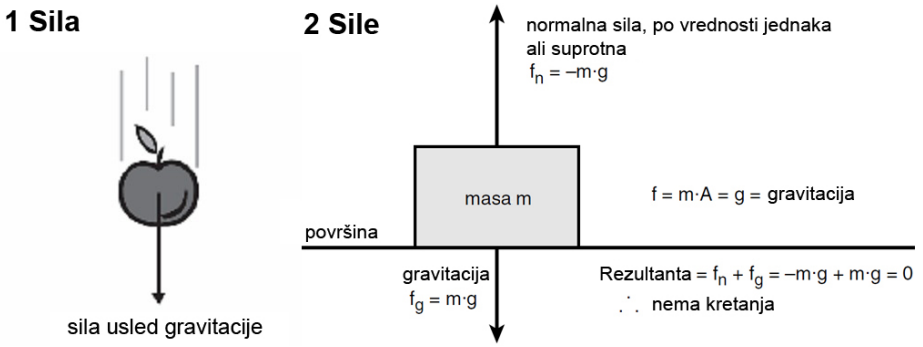
Po definiciji, sila od jednog Njutna je ona koja masi od jednog grama saopštava ubrzanje od 1 metra u sekundi na kvadrat:

$1N = 1g\ 1m/s^2$

Ako na telo deluje više sila F_1 , F_2 , ... F_n , tada se njihova suma naziva *rezultanta*.

Pri dejstvu više sila *II Njutnov zakon* glasi:

$m\,a = \Sigma_i F_i$



Slika 4.1 a) Telo na koje deluje jedna sila, b)Telo na koje deluju dve sile [2]

Sile mogu biti aktivne ili pasivne, što je u vezi sa jednim od Njutnovih zakona, *Zakonom akcije i reakcije*.

Naime, pri svakom dejstvu jednog tela na drugo nekom silom (akcija), drugo telo će delovati na prvo silom istog inteziteta i suprotnog smera.

Otuda, na Slici-1b, na telo deluje sila težine (zemljine gravitacije), a suprotstavlja joj se sila reakcije istog pravca i suprotnog inteziteta.

DRUGI NJUTNOV ZAKON U 2D PROSTORU

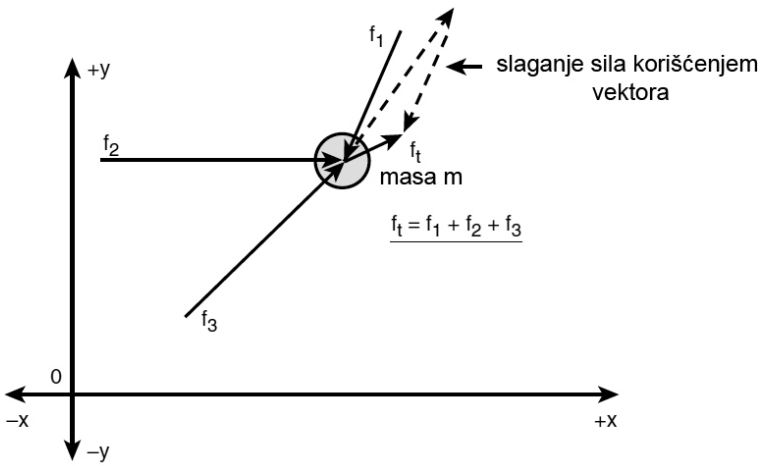
Ubrzanje tela je proporcionalno sumi sila koje deluju na njega

Sila može, naravno, da deluje u sve tri dimenzije a ne samo u pravcu jedne linije. Pogledajmo npr. Sliku-3 na kojoj su naznačene 3 sile koje deluju na materijalnu tačku mase m u 2D ravni. Rezultujuća sila koja deluje na tačku je jednostavno suma svih sila koje deluju na tu tačku. Medjutim, pošto je sila vektorska veličina, neophodno je vektor razložiti na x, z i z komponentu i zatim sabrati svaku od komponenti.

U primeru sa Slike-3 imamo tri sile F_1, F_2, i F_3. Rezultujuća sila F_{final}=<fx, fy> koja dejstvuje na tačku p je jednostavno suma ove tri sile:

$fx = f_{1x} + f_{2x} + f_{3x}$

$fy = f_{1y} + f_{2y} + f_{3y}$



Slika 4.2 Ubrzanje je proporcionalno rezultanti svih sila [2]

U opštem slučaju, rezultujuća sila koja deluje na neki objekat je suma vektora, odnosno matematički:

$F_{final} = F_1+F_2+\ldots+F_n$

gde svaka od sila F_i može da ima 1,2 ili 3 komponente, tj. svaki vektor može da bude 1D(scalar), 2D, ili 3D.

Formulacija II Njutnovog zakona ostaje ista kao i u jednodimenzionalnom slučaju:

Ubrzanje tela je proporcionalno sumi sila koje deluju na njega

$m\,\boldsymbol{a} = \Sigma_i\,\boldsymbol{F}_i$

jedino što u matematičkim relacijama koristimo vektore.

Prethodna vektorska jednačina se može napisati kao dve skalarne jednačine, koje mogu biti praktičnije za primenu tokom implementacije.

$m\,a_x = \Sigma_i\,F_{ix}$

$m\,a_y = \Sigma_i\,F_{iy}$

pri čemu su F_{ix}, F_{iy} komponente sile $\boldsymbol{F}_i$ u pravcu osa x i y.

DIFERENCIJALNE JEDNAČINE KRETANJA

Diferencijalne jednačine su one koje sadrže izvode napoznatih veličina

Napišimo sledeći set relacija koje smo već objasnili u prethodnom tekstu:

$$\frac{\mathrm{d}V_x}{\mathrm{d}t} = (\sum_i^n F_{ix})/m$$

$$\frac{\mathrm{d}V_y}{\mathrm{d}t} = (\sum_i^n F_{iy})/m$$

$$\frac{\mathrm{d}x}{\mathrm{d}t} = V_x$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = V_y$$

U igri i simulaciji računamo u svakom frejmu veličine x, y, V_x, V_y koje se nalaze pod znakom izvoda u gornjim jednačinama, koje se stoga nazivaju *diferencijalne jednačine*.

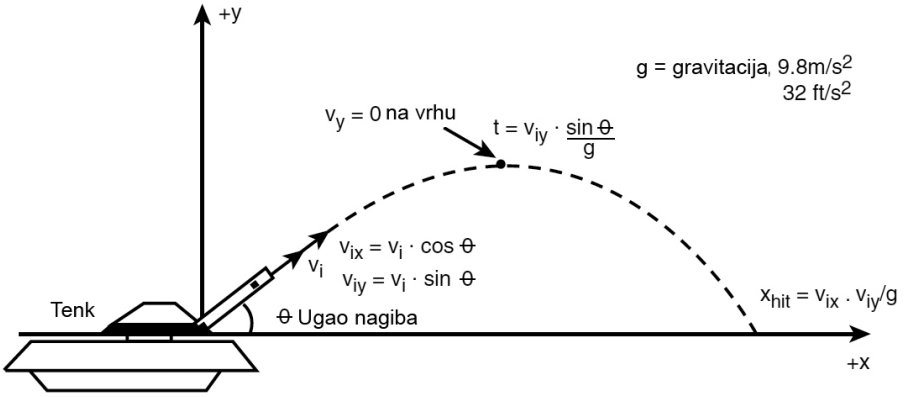
Postupak ažuriranja pozicija i brzina koji smo izložili je u stvari numeričko rešavanje diferencijalnih jednačina i to jednom od najprostijih metoda – Ojlerovom metodom.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PRIMENA II NJUTNOVOG ZAKONA NA KRETANJE PROJEKTILA

Problem kretanja projektila, tj. kosog hica je uobičajen problem u fizici a i u video igrama

Razmotrimo sada primenu izložene materije na problem koji se često sreće u igrama [1]. Na Slici-4 je prikazana postavka problema.



Slika 4.3 Kretanje projektila (kosi hitac) [2]

Imamo zemljino tle, koje ćemo označiti sa $y = O$

i tenk koji je smešten u tački:

$x = 0, y = 0.$

Cev topa je nagnuta prema horizontu pod uglom Θ.

Masa projektila je m a početna brzina V_i (skalarna veličina).

Traži se putanja projektila, odnosno pozicija u svakom frejmu.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

REŠENJE PRIMERA KRETANJA PROJEKTILA

Problem rešavamo tako što najpre razložimo vektor brzine na komponente u pravcu x i y ose

Problem rešavamo tako što najpre razložimo vektor brzine na komponente u pravcu x i y ose. Imamo:

$V_{ix} = V_i * \cos \Theta$

$V_{iy} = V_i * \sin \Theta$

Sada treba na trenutak zaboraviti kretanje u pravcu ose x, i skoncentrisati se na kretanje u pravcu ose y. Jednačina kretanja u pravcu ose z ima sledeći oblik:

$$\frac{\mathrm{d}V_x}{\mathrm{d}t} = (-mg)/m = -g$$

Ubrzanje je: g = 9.81m/s^2, odakle dobijamo da je V_y u proizvoljnom trenutku t:

$V_y\ (t) = V_{iy} - g\ t$

Kad je projektil u najvišoj tački, prestaće da se penje tj. biće $V_y\ (t) = 0$, odakle možemo izračunati vreme uspona projektila:

$t_u = V_{iy} / g = V_i * \sin \Theta / g$

Pri padu će projektilu trebati isto vreme (i postići će vertikalnu brzinu V_{iy} u trenutku udara), pa je vreme leta projektila jednako $2*t_u$.

Kako je u horizontalnom pravcu brzina konstantna (jer smo zanemarili otpor vazduha) to je pozicija projektila na x osi u svakom trenutku t jednaka

$X(t) = V_{ix} * t$

Domet, ili tačka u kojoj će pasti projektil je (stavljajući $t = 2*t_u$) jednaka:

$D = V_{ix}*2*t_u = V_i* \cos \Theta * 2 * V_i*\sin \Theta / g$

$= V_i^2 * (\sin 2\Theta) / g$

Ne izgleda previše komplikovano?

U igri cilj ne mora da bude na istoj visini kao i tenk, tj. na y = 0. U datom slučaju će nam trebati pozicija projektila u svakom frejmu.

Kao što vidimo fizika je u pozadini svega. Ali, kako sve to sada pretočiti u kod? Pa, jedino što treba da uradite jeste da primenite konstantnu brzinu u pravcu ose x i ubrzanje u pravcu ose z, i zatim proveriti kada će projektil udariti u zemlju ili neki drugi objekat. Naravno, u realnom životu će brzine u pravcu X i Y zavisiti od otpora vazduha, ali i bez toga prethodni algoritam radi dovoljno realistično.

KOD ZA KRETANJE PROJEKTILA

Kod za kretanje projektila je dat na sledećem listingu. Često se pretpostavlja da je sila otpora vazduha konstantna

Sledeći listing prikazuje kako se mogu izvršiti potrebna sračunavanja.

```
/// ---------    Inputs
float x_pos     = 0,   // starting point of projectile
y_pos   = SCREEN_BOTTOM, // bottom of screen
y_velocity = 0,   // initial y velocity
x_velocity = 0,   // constant x velocity
gravity = 1,   // do want to fall too fast
velocity   = INITIAL_VEL,  // whatever
angle   = INITIAL_ANGLE; // whatever, must be in radians
// compute velocities in x,y
x_velocity = velocity*cos(angle);
y_velocity = velocity*sin(angle);
// do projectile loop until object hits
```

```
// bottom of screen at SCREEN_BOTTOM
while(y_pos < SCREEN_BOTTOM)
{
    // update position
    x_pos+=x_velocity;
    y_pos+=y_velocity;
    // update velocity
    y_velocity+=gravity;
}       // end while
```

Prethodni kod zanemaruje otpor vazduha i zato će domet tenka biti optimističan. Modeliranje otpora vazduha nije previše komplikovano, ali u jako kompleksnoj igri može biti kap koja će prepuniti čašu.

Otuda, često se pretpostavlja da je sila otpora vazduha konstantna (iako smo videli da zavisi od kvadrata brzine), što daje konstantno ubrzanje duž ose x. Otuda, u prethodni kod se može dodati sledeća linija:

```
x_velocity -= wind_factor;
```

pri čemu se može usvojiti *wind_factor = 0.01* (ili slično) i podestiti tokom testiranja da bi efekat simulacije bio što verniji. Na Slici-5 je dat prikaz jednog frejma na ekranu.

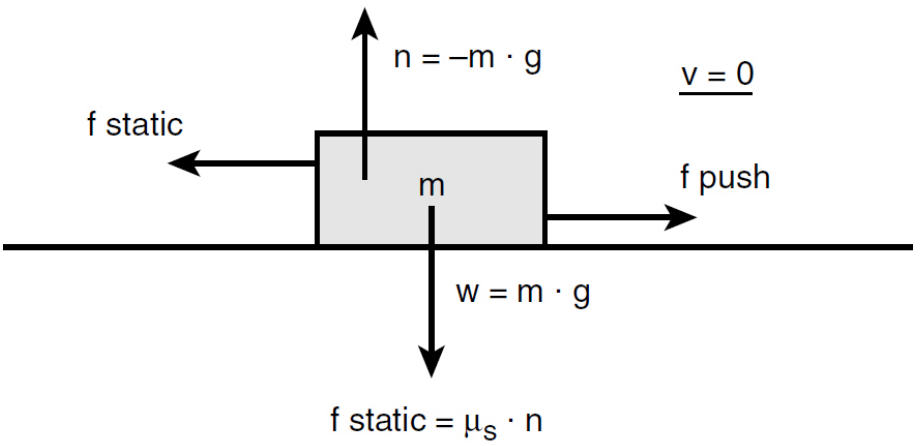
Kompletan projekat je u DeljivimResursima, kao *L12-Cannon.zip*.

SILA TRENJA

Trenje se manifestuje kao otpor kretanju tela i može se predstaviti silom koja deluje suprotno od pravca kretanja

Započenimo sa trenjem i pratimo prethodno definisane korake. Trenje je disipativna (rasipajuća ili usporavajuća). Disipativna sila je ona kod koje se ukupna energija sistema umanjuje kada se objekat kreće.

Trenje se manifestuje kao otpor kretanju tela i može se predstaviti silom koja deluje suprotno od pravca kretanja. Posmatrajmo Sliku-1.



Slika 4.4 Sile na telu pri klizanju, staticka ravnoteza: nema kretanja [2]

Telo mase *m* se nalazi na horizontanoj podlozi u stanju mirovanja (v=0). Na telo deluju sila teže:

$w = mg$

i reakcija podloge:

$n = -mg$.

Ako se na telo deluje nekom horizontalnom silom f_{push}, tada će se javiti suprotna sila koja je jednaka

$f_{static} = \mu_s \bullet m \bullet g$

pri čemu je μ_s *statički koeficijent trenja* (statički zbog stanja mirovanja), koji zavisi od vrste kontaktnih podloga (guma po asfaltu nema isti koeficijent trenja kao skija po snegu) a ne zavisi od površine kontakta.

Zavisno od toga kako se sračunava sila trenja razlikuju se :

- suvo trenje
- viskozno trenje
- otpor vaduha (aerodinamičko trenje) i tecnosti.

Poglavlje 5

Programiranje sila u 2D

UVODNA RAZMATRANJA

Naš cilj je da primenimo silu na posmatrani objekat, bilo da je u pitanju sila vetra ili gravitacije

Već smo spomenuli Drugi Njutnov zakon koji glasi: Sila je jednaka masa puta ubrzanje.

Ali, šta je u stvari masa? Da bi počeli od najprostijih stvari, pretpostavimo da u našem zamišljenom svetu sa pikselima svi objekti imaju masu jednaku 1. F/ 1 = F. Tako imamo da je:

$$\vec{a} = \vec{F}$$

Ubrzanje objekta će u tom slučaju biti jednako sili. U prethodnoj lekciji (L11) smo videli da je ubrzanje bilo ključ za kontrolisanje pomeranja objekata na ekranu. Položaj je zavisio od brzine, a brzina od ubrzanja. Sa ubrzanjem je sve počelo. Sada vidimo da je ustvari sila ta odakle sve počinje.

Vratimo se na našu klasu *Mover* koju smo obradili u prethodnoj lekciji, i koja ima podatke *location*, *velocity*, i *acceleration*.

```
class Mover
{
public:
    PVector location;
    PVector velocity;
    PVector acceleration;
}
```

Naš naredni cilj je da primenimo silu na posmatrani objekat, u obliku:

```
mover.applyForce(wind);
```

ili:

```
mover.applyForce(gravity);
```

gde su vetar (wind) i gravitacija (gravity) podaci tipa *PVector*. Prema Drugom Njutnovom zakonu, ovu funkciju možemo implementirati na sledeći način:

```
void applyForce(PVector force) {
    // Newton's second law at its simplest.
```

```
    acceleration = force;
}
```

Ovo izgleda prilično lepo. Nakon svega, *acceleration = force*, je doslovni prevod Drugog Njutnovog zakona (ako uzmemo da je masa jednaka 1). Pa ipak, postoji veoma veliki problem ovde. Vratimo se na ono što smo ranije hteli da postignemo: kreiranje pokretnog objekta na sceni koji se kreće usled vetra i gravitacije.

```
mover.applyForce(wind);
mover.applyForce(gravity);
mover.update();
mover.display();
```

REZULTUJUĆA SILA

Implementacija rezultujuće sile se vrši u procesu koji je poznat kao slaganje sila

Prethodni kod nije dobar jer je ubrzanje postavljeno na *gravity*, ali to nije ono što smo želeli. Naime, u funkciju *update()* se ubrzanje dodaje na brzinu.

```
velocity.add(acceleration);
```

Vrednost ubrzanja je jednaka sili gravitacije. Vetar je potpuno izostavljen. Postavlja se pitanje kako da savladamo više od jedne sile? Naravno, primenjujemo koncept rezultujuće sile: *Rezultujuća sila je jednaka proizvodu mase i ubrzanja*.

Implementacija ove definicije se vrši u procesu koji je poznat kao *slaganje sila*. Proces je veoma prost; jedino što treba da uradimo je da saberemo sve sile zajedno. U nekom datom vremenskom trenutku, na telo može da deluje 1, 2, 6, 12, ili 303 sila. Dokle god objekat zna kako da odredi rezultujuću silu, nije bitno koliko ima sila koje deluju na njega.

```
void applyForce(PVector force) {
    // Newton's second law, but with force accumulation.
    // We now add each force to acceleration, one at a time.
    acceleration.add(force);
}
```

Slaganje sila ima još jedan deo. Pošto sumiramo sve sile u jednu, u nekom datom vremenskom trenutku, moramo da budemo sigurni da smo obrisali (unulili) ubrzanje svaki put pre poziva funkcije *update()*.

Razmišljajmo sada o vetru. Ponekad je vetar veoma jak, nekada je veoma slab, a neka ga nema uopšte. U nekom posmatranom trenutku, može da se javi veliki nalet vetra, kada korisnik pritisne levi taster miša.

```
void mouse(int button, int state, int x, int y)
{
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
```

```
    {
        PVector wind = new PVector(0.5,0);
        mover.applyForce(wind);
    }
}
```

Kada korisnik pusti taster miša, vetar će prestati, i prema Prvom Njutnovom zakonu, objekat će nastaviti da se kreće konstantnom brzinom. Međutim, ako smo zaboravili da resetujem ubrzanje na 0, jak nalet vetra će i dalje uticati na kretanje. Što je još i gore, biće izvršeno sabiranje sa onim iz prethodnog frejma, s obzirom da se vrši slaganje sila. Najlakši način da implementiramo brisanje ubrzanja u svakom frejmu je da vektor tipa *PVector* pomnožimo sa 0 na kraju svakog poziva funkcije *update()*.

```
void update() {
    velocity.add(acceleration);
    location.add(velocity);
    acceleration.mult(0);
}
```

RAD SA MASOM OBJEKTA

Uključivanje mase je lako kao i dodavanje nove promenljive članice u našu klasu.

Nakon svega, Drugi Njutnov zakon glasi $\vec{F} = m \cdot \vec{a}$ a ne $\vec{a} = \vec{F}$.

Uključivanje mase je lako kao i dodavanje nove promenljive članice u našu klasu. Za početak, dodajemo promenjljivu za masu u klasu *Mover*.

```
class Mover
{
public:
    PVector location;
    PVector velocity;
    PVector acceleration;
    float mass; // Adding mass as a float
```

Masa je skalar (float), ne vektor, tj to je jedan broj koji opisuje količinu materije nekog objekta. Možemo da sračunamo masu tako što ćemo da odredimo površinu objekta i da to posmatramo kao masu. Međutim, započećemo rad tako što ćemo postaviti da je masa objekta na primer, uvek jednaka vrednosti 10.

```
Mover() {
        location = PVector(random(width),random(height));
        velocity = PVector(0,0);
        acceleration = PVector(0,0);
        mass = 10.0;
    }
```

Ovo nije baš najbolje rešenje jer stvari postaju mnogo interesantnije ako imamo objekte sa promenljivom masom. Kako onda uključujemo masu u kod?

Naime, mi je koristimo dok primenjujemo Drugi Njutnov zakon na naš objekat.

```
void applyForce(Pvector force) {
        // Newton's second law (with force accumulation and mass)
        force.div(mass);
        acceleration.add(force);
    }
```

PRIMER: OBJEKTI SA PROMENLJIVOM MASOM

Uključujemo više objekata sa promenljivom masom i primenjujemo klase kako bi primer načinili još interesantnijim

Pogledajmo sada kako bi smo ovaj primer načinili još interesantnijim, tako što bi smo uključili više objekata sa promenljivom masom. Da bi smo ovo uradili, primenićemo klase i princip OO programiranja.

Primetimo sličnosti sledeće klase sa već kreiranom klasom *Mover* koju smo imali ranije. Jedina dva dodatka su— *mass* i nova *applyForce()* funkcija.

```
class Mover
{
public:
    PVector location;
    PVector velocity;
    PVector acceleration;
    float mass;         //  The object now has mass!

    Mover() {
        //  And for now, we'll just set the mass equal  to 1 for simplicity.
        mass = 1;
        location = PVector(30,30);
        velocity = PVector(0,0);
        acceleration = PVector(0,0);
    }

    void applyForce(PVector force) { // Newton's second law.
        // Receive a force, divide by mass, and add to acceleration.
        PVector f = force;
        f.div(mass);
        acceleration.add(f);
    }

    void update() {
        velocity.add(acceleration);  // Motion 101 from Chapter 1
        location.add(velocity);
        // Now add clearing the acceleration each time!
```

```
        acceleration.mult(0);
    }

    void display() {
        stroke(0);
        fill(175);
        // Scaling the size according to mass.
ellipse(location.x,location.y,mass*16,mass*16);
    }

    void checkEdges()
    {
        // When it reaches one edge, set location to the other.
        if (location.x > width) {
            location.x = 0;
        } else if (location.x < 0) {
            location.x = width;
        }
        if (location.y > height) {
            location.y = 0.0;
        } else if (location.y < 0) {
            location.y = height;
        }
    }
};
```

GENERISANJE POKRETNIH OBJEKATA

Kada imamo definisanu klasu Mover možemo da kreiramo stotinu objekata i simuliramo kretanje

Sada kada smo definisali našu klasu, možemo da kreiramo, na primer, stotinu objekata tipa *Mover* u okviru jednog niza.

```
Mover movers[100];
```

a zatim možemo da inicijalizujemo sve ove objekte tipa Mover u funkciji *setup()* korišćenjem petlje.

```
void setup()
{
    size(200,200);
    setTimer(30);

    for (int i = 0; i < 100; i++) {
        movers[i] = Mover();
    }
}
```

Ali sada imamo jedan mali problem. Ako se vratimo na konstruktor klase *Mover*

```
Mover() {
        //  And for now, we'll just set the mass equal  to 1 for simplicity.
        mass = 1;
        location = PVector(30,30);
        velocity = PVector(0,0);
        acceleration = PVector(0,0);
    }
```

vidimo da je svaki objekat klase *Mover* kreiran na identičan način.

Ono što želimo ovde je da imamo *Mover* objekte različite mase koji započinju kretanje sa različitih lokacija. Stoga je potrebno da uključimo i konstruktore sa parametrima.

```
Mover(float m, float x , float y) {
        mass = m;  // Now setting these variables with arguments
        location = PVector(x,y);
        velocity = PVector(0,0);
        acceleration = PVector(0,0);
    }
```

Na ovaj način smo izbegli hardkodiranje, i elagantnije, prosledili vrednosti korišćenjme konstruktora. Ovo znači da sada možemo da kreiramo puno *Mover* objekata: velike, male, one koji počinju kretanje na levoj strani ekrana, oni koji započinju kretanje na desnoj strani, itd.

```
Mover m1 (10,0,height/2);       //  A big Mover on the left side of the window
Mover m1 (0.1,width,height/2);  // A small Mover on the right side of the window
```

Ono što želimo je da u okviru niza inicijalizujemo sve objekte korišćenjem petlje:

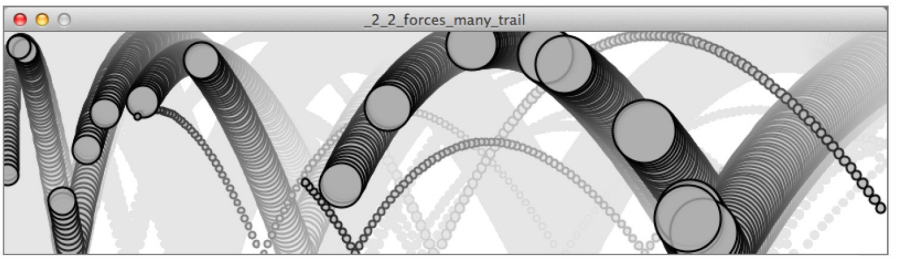
```
void setup()
{
    setTimer(30);

    for (int i = 0; i < 100; i++) {
        // Initializing many Mover objects, all with
        // random mass (and all starting at 0,0)
        movers[i] = Mover(random(0.1,5),0,0);
    }
}
```

ISCRTAVANJE POKRETNIH OBJEKATA

Za svaki kreirani pokretni objekat, masa je postavljena na slučajnu vrednost izmedju 0.1 i 5, početna x lokacija je postavljena na 0, a takodje i početna y lokacija

Za svaki kreirani pokretni objekat, masa je postavljena na slučajnu vrednost izmedju 0.1 i 5, početna x lokacija je postavljena na 0, takodje i početna y lokacija . Očigledno, postoji

ogroman broj načina koje možemo da izaberemo za inicijalizaciju objekata. Jednom kad je niz objekata deklarisan, kreiran i inicijalizovan, ostatak koda je prilično prost..



Slika 5.1 Primer kretanja većeg broja objekata različitih dimenzija i masa [4]

```
void draw() {
    background(255);
    PVector wind(0.01,0);
    PVector gravity(0,0.1);   //  Make up two forces.

    // Loop through all objects and apply both forces to each object.
    for (int i = 0; i < 100; i++) {
        movers[i].applyForce(wind);
        movers[i].applyForce(gravity);
        movers[i].update();
        movers[i].display();
        movers[i].checkEdges();
    }
}
```

Primetimo kako u prethodnom primeru, najmanji kružić je dostigao desnu ivicu brže od većih. To je zbog same formule:

ubrzanje = sila / masa.

Što je veća masa to je manje ubrzanje.

Poglavlje 6

Vežbe

PRIMER: KONSTANTNO UBRZANJE

U klasu Mover dodajemo još jednu promenljivu tipa PVector - acceleration.

Konstantno ubrzanje nije nešto naročito zanimljivo, ali je najprostiji oblik, i pomoćiće nam da ugradimo osnove ubrzanje u naš kod. Prva stvar koju treba da uradimo je da dodamo još jednu promenljivu tipa *PVector* u našu *Mover* klasu:

```
class Mover
{
public:
    PVector location;
    PVector velocity;
    PVector acceleration; // A new PVector for acceleration
```

Odgovarajuće uključivanje ubrzanja u funkciji *update()* bi bilo:

```
void update() {
        // Our motion algorithm is now two lines of  code!
        velocity.add(acceleration * dt);
        location.add(velocity * dt);
    }
```

Skoro smo završili. Jedina preostala stvar je inicijalizacija vrednosti u konstrukoru.

```
Mover() {
```

Neka objekat Mover započinje kretanje iz centra prozora:

```
location = new PVector(width/2,height/2);
```

inicijalnom brzinom jednakom nuli.

```
velocity = PVector(0,0);
```

Ovo znači da kada započne rad aplikacije, objekat je u mirovanju. O brzini dalje ne moramo da vodimo računa, i kretanje objekta kontrolišemo isključivo korišćenjem ubrzanja. Drugim rečima, naš prvi kod uključuje konstantno ubrzanje.

```
acceleration = PVector(-0.001,0.01);
}
```

Možda sada pomislite *"Pa ove vrednosti su previše male!"* U pravu ste, vrednosti su baš male. Ali, bitno je shvatiti da se vrednosti ubrzanja, mereno u pikselima, akumuliraju u promenljivoj *velocity* u toku vremena, u proseku oko 30 puta u jednoj sekundi, u zavisnosti od stope osvežavanja koju propišemo. I stoga, da bi smo vrednost brzine držali u nekom razumnom opsegu, vrednosti ubrzanja treba da ostanu veoma male. Da bi smo ovo sprečili možemo u klasu *PVector* da dodamo funkciju *limit()*.

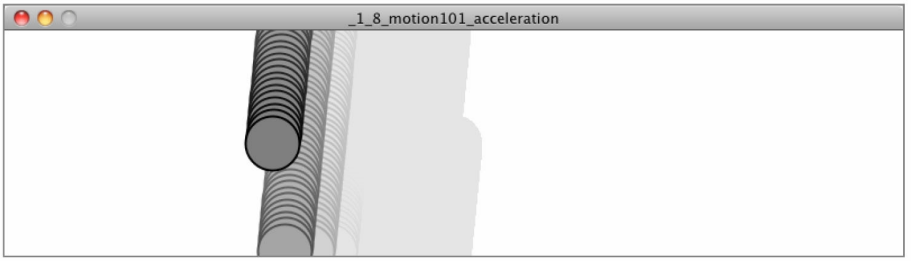
```
// F limit() function constrains
// the magnitude of a vector.
velocity.limit(10);
```

Ovo se prevodi na sledeće: Kolika je vrednost (intenzitet) brzine? Ukoliko je manja od 10 onda je sve u redu; ostavi je kakva jesta. Ali ako je veća od 10 onda je ograničite na vrednost 10!

KONSTANTNO UBRZANJE - KOD

Vrednosti ubrzanja, mereno u pikselima, se akumuliraju u promenljivoj velocity u toku vremena

Pogledajmo sada izmene u klasi *Mover*, nakon dodavanja promenljive *acceleration* i primene funkcije *limit()*.



Slika 6.1 Primer sa konstantnim ubrzanjem objekta [6]

```
// L11 - Primer 2: Kretanje 101 (brzina i konstantno ubrzanje)

class Mover
{
public:
    PVector location;
    PVector velocity;
    PVector acceleration; // A new PVector for acceleration

    // The variable topspeed will limit the magnitude of velocity.
    float topspeed;

    Mover() {
        location = PVector(width/2.,height/2.);
```

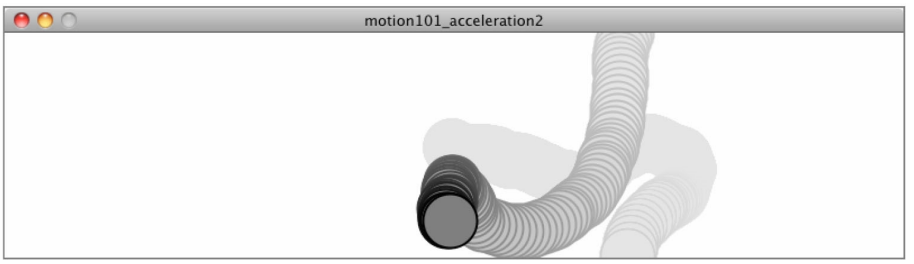
```
        velocity = PVector(0,0);
        acceleration = PVector(-0.001,0.01);
        topspeed = 10;
    }

    void update() {
        velocity.add(acceleration * dt);
        velocity.limit(topspeed); // Velocity changes by acceleration and is
limited by topspeed.
        location.add(velocity * dt);
    }

     void display() {}  // display() is the same.

     void checkEdges() {} // checkEdges() is the same.
}
```

PRIMER: POTPUNO SLUČAJNO UBRZANJE

U svakom sledećem frejmu se zadaje nova i slučajno izabrana vrednost ubrzanja

U ovom slučaju, umesto inicijalizacije vrednosti ubrzanja u konstruktoru objekta, mi želimo da izaberemo novo ubrzanje u svakom sledećem ciklusu, tj svaki put kada pozovemo funkciju update().

```
// L11 - Primer 3: Kretanje 101 (brzina i proizvoljno ubrzanje)

    void update()
    {
        // The random2D() function will give us a PVector of length 1
                // pointing in a random direction.
        acceleration.random2D();
        velocity.add(acceleration * dt);
        velocity.limit(topspeed);
        location.add(velocity * dt);
    }
```



Slika 6.2 Primer sa proizvoljnim i slučajnim ubrzanjem [6]

Pošto je slučajan vektor jedinični (tj, već je normiran) možemo da probamo da ga skaliramo:

(a) skaliranje ubrzanja korišćenjem konstantne vrednosti

```
acceleration = PVector.random2D();
acceleration.mult(0.5); // Constant
```

(b) skaliranje ubrzanja korišćenjem slučajne (nasumične) vrednosti

```
acceleration = PVector.random2D();
acceleration.mult(random(2)); // random
```

Iako ovo može delovati kao očigledna stvar, veoma je bitno razumeti da se ubrzanje ne odnosi samo na ubrzanje ili usporenje kretanja objekta, već i na promenu intenziteta i pravca brzine objekta..
Vežbanje 1.4
Pokušajte sami da napišete funkciju limit() za PVector klasu.
void limit(float max) {
if (_____ > _____) {
_____();
____(max);
}
}

DEMO: HELIHOPTER

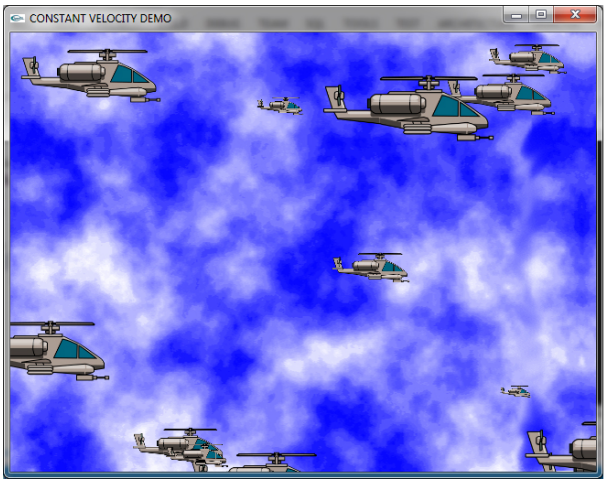
Demo Chopper pokazuje flotu helikoptera koji se kreću konstantnom brzinom

U okviru sekcije Deljivi materijali se nalazi *L11-ChopDemo.zip* koji pokazuje flotu helikoptera koji se kreću konstantnom brzinom, i koji je realizovan u GLUT / OpenGL / C++ okruzenju, Slika-7.

Projekat koristi SOIL biblioteku za učitavanje slike, i FMOD za generisanje zvuka. Promenljive u igri su:

```
//Game variables
Sprite* chopers[NUM_CHOPPERS];
Sprite* background[NUM_BACKGR];
```

pri čemu je NUM_CHOPPERS setovano na 24, a NUM_BACKGR na 2.



Slika 6.3 Demonstracija programa sa helihopterima, modifikovano prema [1]

Pozadina se pomera s desno na levo. Da bi slika bila kontinualna uvodimo dve pozadine (kada je jedna pozadina delimično van ekrana, tu crninu prekriva druga pozadina koja se takođe kreće s leva na desno). Kada jedna pozadina izađe potpuno sa ekrana pojavljuje se na drugoj strani ekrana. Slično važi i za helihoptere. Oni se kreću sa desna na levo, kada pređu desnu granicu ekrana pojavjuju se na leveoj strani ekrana.

U funkciji *LoadTextures()* se učitavaju teksture za pozadine i helihoptere i postavljaju početne pozicije, i inicijalizuju veličine i brzine svakog pojedinačnog helihoptera. Ažuriranje objekata se kao i do sada vrši u funkciji *Update()* klase *Sprite*.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

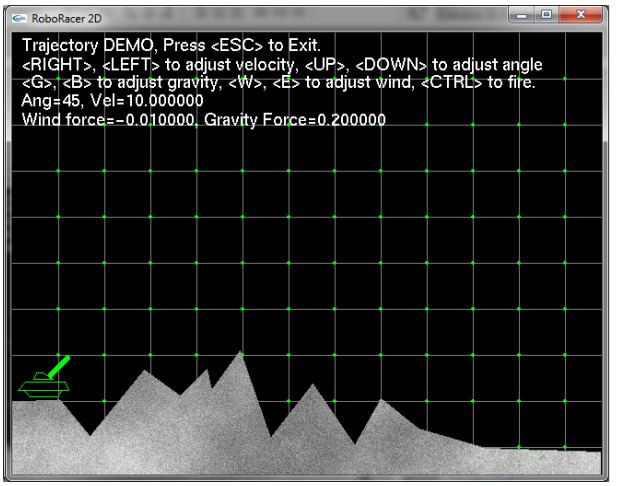
DEMO: CANNON

Kod za kretanje projektila je dat na sledećem listingu. Često se pretpostavlja da je sila otpora vazduha konstantna

Kod opisan na predavanjima omogućama generisanje tenka, i usmeravanja njegovog topa, a zatim i ispaljivanje projektila. Kompletan projekat je u fajlu *L12-Cannon.zip*.

Lista projekatila se cuva u nizu:

```
PROJECTILE missiles[NUM_PROJECTILES];
```



Slika 6.4 Izgled ekrana igre iz primera [2]

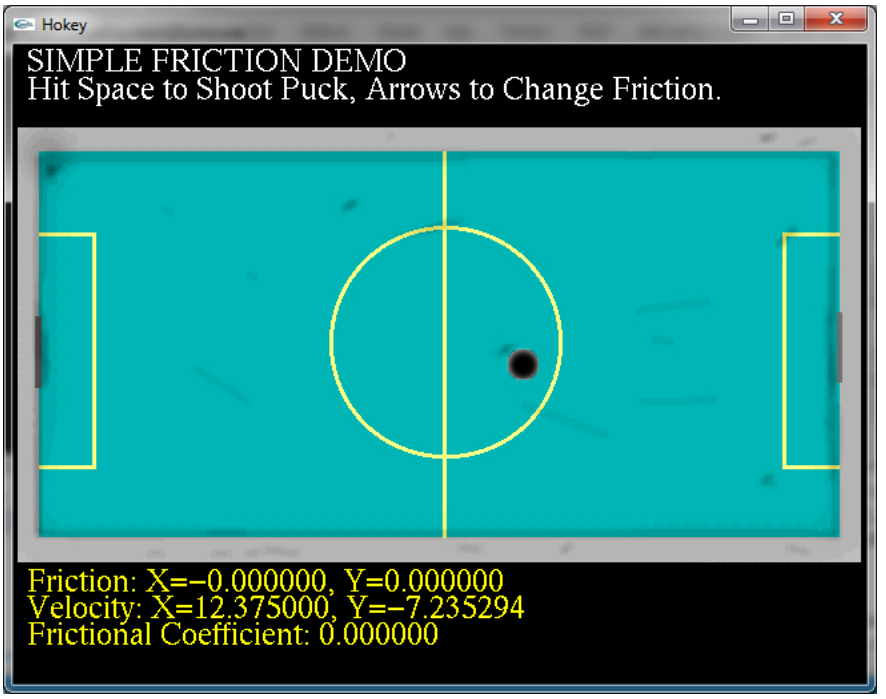
U ekranu igre su prikazane kontrole koje omogucavaju promenu ugla, topa, povecanje i smanjenje sile gravitacije i otpora vazduha, kao i ispaljivanje projektila.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

DEMO: HOKEY

Program omogućava da pritiskom na Space ispalite lopticu. Koeficijent trenja se menja strelicama levo /desno

Primer upotrebe trenja je air hockey demo, tj *L12-Hockey.zip*. Program omogućava da ispalite pak u slučajnom pravcu, svaki put kada pritisnite taster Space. Pak zatim udara u ivice terena i odbija se nazad. Istovremeno, na pak deluje sila trenja koja dovodi pak do zaustavljanja.



Slika 6.5 Hokey Demo [2]

Korišćenjem strelica možete da menjate koeficijent trenja. Pokušate da dodate palice u projekat koje će sprečiti postizanje gola, i koje će biti vođenje od strane računara.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Kod za demo Cannon i Hokey se može preuzeti sa sledećeg linka:

https://drive.google.com/file/d/1O3V2Ixe0v7mdfmrrlSmbdLWGTkiM-qGu/view?usp=sharing

Poglavlje 7

Individualne vežbe

ZADACI ZA SAMOSTALAN RAD

Na osnovu materijala uraditi samostalno sledeće zadatke

Zad 1. Kreirati simulaciju automobila (ili trkača) koji ubrzava kada pritisnete dugme up na tastaturi a usporava (koči) kada pritisnete taster down. (15 minuta)

Zad 2. Preuzeti **L11-ChopDemo.zip**.

- Omogućiti da helihopter ima i vertikalnu komponentu pomeranja. Kada pridje previše blizu zemlje odbija se od nje.
- Omogućiti slučajno ubrzanje u oba pravca kretanja (x,y)
- Preuzeti sa neta drugu pozadinu i druge listove sa sprajtovima (helihopter, leptir, ptica), i primeniti ih u programu.

(30 minuta)

Zad 3. Preuzeti projekat **L11-OpenGL.zip** iz dodatnih materijala lekcije L11.

Analizirati sekciju 5.1 Programiranje kretanja u 2D, lekcije L11. Gotova klasa Mover je već implementirana u kodu koji ste preuzeli, i sadrži promenljive location, velocity i acceleration koje omogućavaju kontrolisanje ravnomernog i ubrzanog kretanja loptica. Fajl main.cpp sadrži određenu funkcionalnost ali ga treba izmeniti da zadovolji zahteve domaćeg zadatka.

- Izmeniti kod fajla main.cpp tako da se lopta kreće konstantnim ubrzanjem kao u primeru PRIMER: KONSTANTNO UBRZANJE, sekcije 5.1. Izvorni kod snimiti kao Z11_A.cpp.
- Izmeniti kod fajla Z11_A.cpp tako da se lopta kreće slučajnim ubrzanjem kao u primeru POTPUNO SLUČAJNO UBRZANJE, sekcije 5.1. Izvorni kod snimiti kao Z11_B.cpp. .

(30 minuta)

Zad 4. Primenom sila, simulirati kretanje balona ispunjenog helijumom koji se penje na gore i odbija se od gornje ivice prozora. Dodajte silu vetra koja se menja u toku vremena, prema nekom zakonu. (10 min)

Zad 5. Umesto da se objekti odbijaju od zida u primeru sa silama, kreirati primer u kome postoji nevidljiva sila koja deluje na objekat tako da mu ne da da izadje iz prozora. Da li možete da primenite silu koja zavisi od toga koliko je objekat udaljen od ivice prozora — tj. što je objekat bliži prozoru to je sila veća. (15 min)

Zad 6. Kreirati kvadratne zone u primeru sa trenjem tako da objekti iskuse silu trenja kada prelaze preko te zone. Šta će se desiti ako varirate jačinu trenja (tj. koeficijent trenja) svake

od površina? Šta će se desiti ako neki od kvadrata proizvodi silu suprotnu sili trenja — tj. kada uđete u datu zonu ona ustvari ubrzava objekat umesto da ga usporava? (15 min)

Zaključak

REZIME

Na osnovu svega obrađenog možemo zaključiti sledeće

U izlaganju je ukazano da je za uspešnu računarsku igru neophodna sličnost sveta igre u kome se radnja dogadja sa realnim svetom, i da ta sličnost ima dve komponente: geometrijsko-grafičku koja se naziva realizam i fizičku koja se naziva vernost.

Vernost prikaza u igri je obezbeđena ako se objekti u igri ponašaju po zakonima fizike, koji važe i u realnom svetu. Proučeno je jednodimenzionalno kretanje i izloženi su osnovni pojmovi kao što su koordinatni sistemi, položaj (pozicija) tela, brzina ubrzanje, kao i merne jedinice.

Pokazano je kako se uvedene relacije koriste tokom simulacije za ažuriranje pozicije i brzine objekta u igri.

Telo se može smatrati materijalnom tačkom ako su njegove dimenzije zanemarivo male u odnosu na rastojanja koja prelazi. Za određivanje njegovog položaja su potrebna dva podatka (npr koordinate x i y). Kažemo da materijalna tačka u ravni ima dva stepena slobode (2DOF – Degrees Of Freedom).

Izložene su relacije sile i ubrzanja, Videli smo da je sila proporcionalna ubrzanju i da je masa faktor proporcionalnosti. Videli smo da su brzina ubrzanje sila vektori i da Drugi Njutnov zakon dobija vektorsku formu.

REFERENCE

Korišćena literatura

[1] Andre Lamothe, **Tricks of the Windows Game Programming Gurus**, Sams, 2002

[2] Wendy Stahler, **Beginning Math and Physics for Game Programmers**, New Riders Publishing, 2004.

[3] http://www.animations.physics.unsw.edu.au/

[4] David Bourg, **Physics for Game Developers**, O'Reilly&Associates, Inc. Sebastopol, 2002

[5] Grant Palmer, **Physics for game programmers**, Springer-Verlag, New York, 2005.

[6] Daniel Shiffman, **The Nature of Code: Simulating Natural Systems with Processing** , 2014.