SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 4858

Fizikalno temeljena simulacija sudaranja kugli

David Emanuel Lukšić

Zagreb, lipanj 2017.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

Ovim putem pohvalio bih organizaciju Fakulteta elektrotehnike i računarstva u Zagrebu. Zahvaljujem se svim profesorima i asistentima, jer su bili profesionalni, spremni pomoći i zainteresirani za svoje studente. Posebno se zahvaljujem svojim roditeljima za financijsku podršku tijekom studija i svojoj mentorici Željki Mihajlović za upute i pomoć pri pisanju ovog rada.

Sadržaj

Popis slika

1. Uvod

Biljar je jedan od popularnijih žanrova u industriji video igara. Neke od tih igara koriste vrlo jednostavne (nerealistične) modele i fokusirane su na "igrivost" i grafiku (npr. "Pool Nation FX", 2012.). Druge pak su više koncentrirane na točnost fizikalnog modela, imaju i vertikalnu slobodu, skokove (npr. "Carom 3D", 1999.). U ovom radu razmotrit ćemo kako implementirati općeniti simulator kugli, a zatim ga upotrijebiti kako bismo simulirali igru biljara.

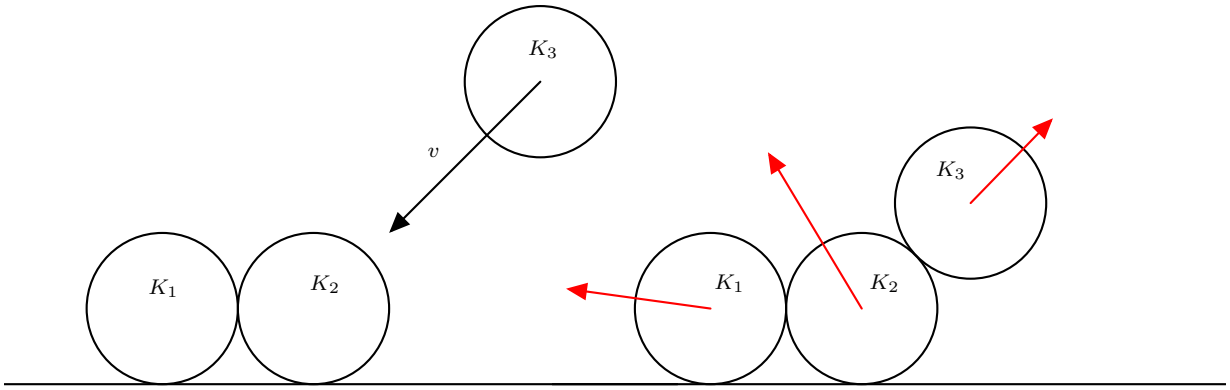
Glavna ideja je da naš matematički model pruža dovoljno parametara kako bismo mogli simulirati neke specifične situacije iz igre biljara, a da pri tom izgledaju realistično. Dakle, cilj nije pronaći točne vrijednosti tih parametara nego omogućiti modeliranje takvih situacija, a da se pri tome one poklapaju sa stvarnošću ukoliko uzmemo povoljne parametre.

Na slici 1.1, kao motivacijski primjer, pokazat ćemo koliko je važna matematička podloga simulatora koji koristimo. Kugle K_1 i K_2 "priljubljene" su uz rub stola i međusobno. Kugla K_3 kreće se prema kugli K_2 pod kutem od 45°. Što će se dogoditi?

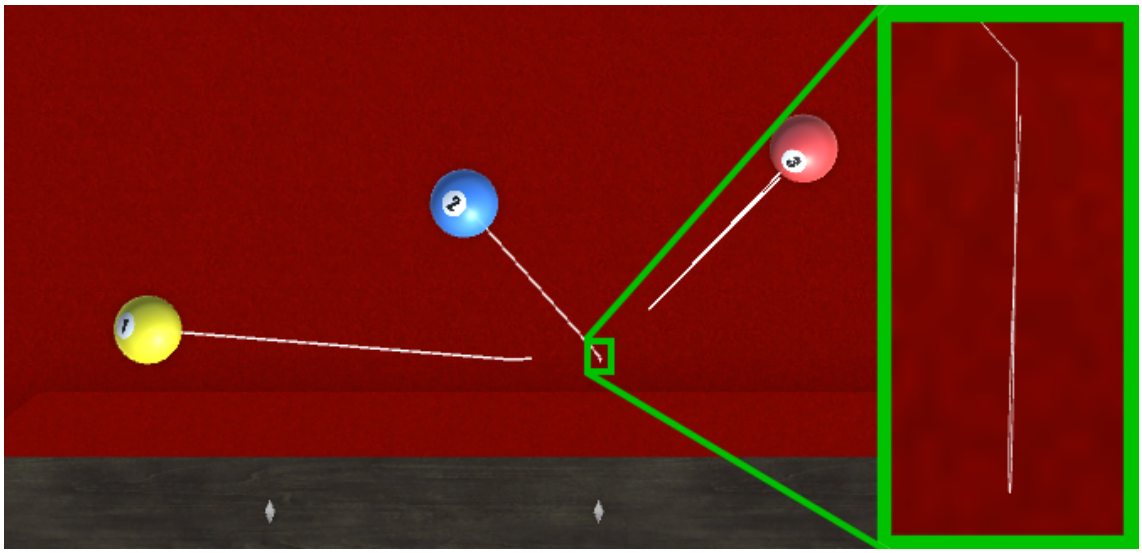
To zapravo i nije sasvim jasno. Naivnim zaključivanjem, došli bismo do zaključka da će K_1 krenuti niz rub stola, K_2 odbiti pod pravim kutem od stola, dok će K_3 krenuti negdje nazad, nekom brzinom, to ne možemo biti sigurni. Naravno pri ovakvim situacijama potrebno je poznavati fizikalne materijale od kojih se pojedini dijelovi simulacije sastoje. Jedna od bitnih informacija jest da je tvrdoća kugle (tvrda plastika) daleko veća nego tvrdoća ruba stola (guma). Zatim, iako se čini zanemarivo, trenje između kugli postoji te iznosi $0.03 - 0.08$. Znajući to dolazimo do neobičnog rješenja (slika 1.2): K_1 će se blago odbiti od ruba stola zbog vertikalne brzine dobivene trenjem s K_2. K_2 će se odbiti od K_3 i saviti ("ući u") rub stola. Zatim će se vratiti, te opet sudariti s K_3 koja se odbije u smjeru otprilike odakle je došla (ne istom smjeru, zbog blagog pomaka K_2 tijekom odbitka od ruba stola!), a K_2 nastavi u smjeru prema naprijed pod nekim kutem manjim od 90°. Čak ni sami sudari kugli nisu trenutni događaji, nego imaju svoje

trajanje, što pak donosi još naizgled neočekivanih ponašanja.

Simulatorom koji je prezentiran u ovom radu, moći ćemo simulirati ovu i mnoge druge zanimljive situacije. Uz to očekujemo simulaciju u stvarnom vremenu i determinizam u odnosu na odabir veličine koraka.



Slika 1.1: Motivacijski primjer



Slika 1.2: Rješenje motivacijskog primjera u simulaciji

2. Odabir matematičkog modela

2.1. Kontinuirani model

Simulirati kretanje kugli kontinuiranim funkcijama čini se vrlo primamljivo. U 3. poglavlju pokazat ćemo kako se kretanje kugli u mnogo slučajeva može prikazati običnim paraboličnim krivuljama: padanje, klizanje, usporavanje. Svi ovi primjeri sadrže konstantne sile, a rezultantna kretanja su parabole. Iz tog razloga, čini se jednostavno simulaciju podijeliti u niz "događaja", odnosno sudara, prilikom kojih se brzine kugli promijene pa se izračunaju novi sudari i tako dalje. Međutim, ovaj pristup zahtijeva korištenje grubih aproksimacija sudara, koje zanemaruju vrlo kompleksne procese pa time gubimo realističnost. Ovakvi sustavi su korisni ako smo ograničeni računalnim resursima (npr. mobilni uređaji), ili nam je potrebno simulirati mnogo situacija za potrebe strojnog učenja.

2.2. Iterativni model

Iterativni modeli baziraju se na rješavanju *običnih diferencijalnih jednadžbi* (ODJ). Najjednostavniji način njihovog rješavanja jest *Eulerova metoda*, koja se često koristi u video igrama. U slučaju kretanja čvrstog tijela Eulerova metoda glasi:

$$\begin{aligned} p_{n+1} &= p_n + v_n dt \\ v_{n+1} &= v_n + a_n dt \end{aligned} \tag{2.1}$$

Gdje nam p_n označava trenutnu poziciju tijela, v_n trenutnu brzinu, a a_n trenutnu izračunatu akceleraciju. Sustav simuliramo u diskretnim koracima $n = 0, 1, 2, ...$ s vremenskim razmakom dt. Eulerova metoda unosi vrlo veliku pogrešku, koja je linearna s korakom dt (obično je $dt < 1$). Razvijene su mnoge druge metode (*Midpoint method*, *Runge-Kutta 4 (RK4)*, *Adams-Bashforth-Moulton*), koje uz malo više računanja imaju polinomnu ($dt^2, dt^3...$) ovisnost pogreške o koraku. Nova, vrlo popularna i robusna metoda *Bulirsch-Stoer* pogodna je za simuliranje

sustava s velikim "skokovima" u rješenju iz razloga što koristi prilagodljivi korak simulacije: na mjestima gdje se rješenje ne mijenja brzo, radi velike korake, a na mjestima gdje se događaju skokovi provede više manjih koraka.

Za potrebe ovog rada koristio sam C++ biblioteku *Odeint*[1] koja sadrži implementacije navedenih i drugih metoda integracije. Svaka metoda zahtijeva definiciju sustava danu s ODJ u sljedećem obliku:

$$\frac{d\bar{\boldsymbol{x}}}{dt} = f(t, \bar{\boldsymbol{x}}) \tag{2.2}$$

gdje nam $\bar{\boldsymbol{x}}$ označava niz varijabli koje sudjeluju u sustavu. U slučaju sustava simulacije kugli, taj niz predstavlja niz kugli od kojih se svaka sastoji od trodimenzionalnih vektora položaja $\vec{p}$, vektora brzine $\vec{v}$ i vektora kutne brzine $\vec{\omega}$:

$$\bar{\boldsymbol{x}} = \begin{Bmatrix} p_{1x}, p_{1y}, p_{1z}, & v_{1x}, v_{1y}, v_{1z}, & \omega_{1x}, \omega_{1y}, \omega_{1z}, \\ p_{2x}, p_{2y}, p_{2z}, & v_{2x}, v_{2y}, v_{2z}, & \omega_{2x}, \omega_{2y}, \omega_{2z}, \\ p_{3x}, p_{3y}, p_{3z}, & \cdots & \\ \vdots & & \end{Bmatrix} \tag{2.3}$$

U slučaju kretanja čvrstog tijela, recimo jedne kugle bez rotacije, s gravitacijom g, pisali bismo:

$$\bar{\boldsymbol{x}} = \begin{Bmatrix} p_{1x}, p_{1y}, p_{1z}, & v_{1x}, v_{1y}, v_{1z} \end{Bmatrix}$$

$$f(t, \bar{\boldsymbol{x}}) = \begin{cases} dp_{1x}(t, \bar{\boldsymbol{x}}) = v_{1x}\, dt \\ dp_{1y}(t, \bar{\boldsymbol{x}}) = v_{1x}\, dt \\ dp_{1z}(t, \bar{\boldsymbol{x}}) = v_{1z}\, dt \\ dv_{1x}(t, \bar{\boldsymbol{x}}) = 0 \\ dv_{1y}(t, \bar{\boldsymbol{x}}) = -g\, dt \\ dv_{1z}(t, \bar{\boldsymbol{x}}) = 0 \end{cases} \tag{2.4}$$

3. Fizikalni model kugle

3.1. Fizikalni parametri

Stanje svake kugle u nekom trenutku opisano je sljedećim varijablama: položaj u prostoru $\vec{p}$, linearna brzina $\vec{v}$ i kutna brzina $\vec{\omega}$. Kutna brzina je vektor čiji smjer označava os rotacije, a iznos brzinu okretanja (rad/s).

Osim tih parametara, kugla ima statične parametre: masu m i radijus R.

3.2. Jednadžbe za jednu kuglu

Kretanje kugle kroz vrijeme opisujemo običnom diferencijalnom jednadžbom drugog reda. Kako Odeint integrator radi samo s diferencijalnim jednadžbama prvog reda, moramo sve jednadžbe napisati pomoću supstitucije na sljedeći način:

$$\begin{aligned} d\vec{p} &= \vec{v}\, dt \\ d\vec{v} &= \vec{a}(t, \bar{\boldsymbol{x}})\, dt \\ d\vec{\omega} &= \vec{\alpha}(t, \bar{\boldsymbol{x}})\, dt \end{aligned} \tag{3.1}$$

gdje su akceleracija $\vec{a}(t, \bar{\boldsymbol{x}})$ i kutna akceleracija $\vec{\alpha}(t, \bar{\boldsymbol{x}})$ funkcije vremena t i stanja sustava $\bar{\boldsymbol{x}}$ u tom trenutku. Ove funkcije računat ćemo tako da zbrojimo sve sile koje djeluju na kuglu. Pretpostavimo da na kuglu djeluje n sila $\vec{F_1}, \vec{F_2}, ..., \vec{F_n}$ na relativnim pozicijama od kugle $\vec{r_1}, \vec{r_2}, ..., \vec{r_n}$:

$$\begin{aligned} \vec{a} &= \sum_{i=1}^{n} \vec{F_i}/m \\ \vec{\alpha} &= \sum_{i=1}^{n} \left(\vec{r_i} \times \vec{F_i}\right) \frac{5}{2mR^2} \end{aligned} \tag{3.2}$$

Možete primjetiti da kroz ove jednadžbe ne pratimo "kutni položaj". To je zato što točan kutni položaj nije bitan za simulaciju (kugla je simetrična oko svog središta). Prikaz kutnog položaja, odnosno rotaciju kugle, aproksimirat ćemo iz vrijednosti kutne brzine koju vraća simulator.

3.3. Račun osnovnih sila

Na svaku kuglu u nekom trenutku djeluju sljedeće sile:

- Sila gravitacije
- Sila trenja s površinom stola (klizanje, kotrljanje)
- Sile sudara

U nekim slučajevima jednostavnije je izraziti promjenu akceleracije $\triangle\vec{a}$ i promjenu kutne akceleracije $\triangle\vec{\alpha}$ koju određena sila uvodi, jer ne ovisi o masi odnosno radijusu.

3.3.1. Sila gravitacije

Sila gravitacije je konstantna i stvara akceleraciju koja je konstantna i ne ovisi o masi:

$$\triangle\vec{a_g} = (0, -g, 0) \tag{3.3}$$

gdje je g gravitacijska konstanta.

3.3.2. Sila trenja s površinom stola

Trenje s površinom stola je malo kompliciranija sila utoliko što ovisi o iznosu i smjeru klizanja točke dodira kugle i stola. Brzinu klizanja dodirne točke računamo ovako:

$$\vec{u} = \vec{v} + R\hat{y} \times \vec{\omega} \tag{3.4}$$

Klizanje kreće kada je $|\vec{u}| > 0$. Sile trenja djeluju u dodirnoj točki. Neka je $\hat{u}$ jedinični vektor u smjeru $\vec{u}$, a $|\vec{F_n}|$ iznos kontaktne sile sa stolom (više o tome u 4. poglavlju):

$$\begin{aligned} \vec{r_{tr}} &= -R\vec{y} \\ \vec{F_{tr}} &= -\mu \left|\vec{F_n}\right| \hat{u} \end{aligned} \tag{3.5}$$

gdje je μ konstanta trenja klizanja. Može se pokazati da $|\hat{u}|$ ne mijenja smjer tijekom klizanja, što znači da se kugla kreće po paraboli [4].

Ukoliko je $|\vec{u}| = 0$, kugla se kotrlja bez klizanja i javlja se trenje kotrljanja zbog deformacije tkanine koja je na stolu. Zbog ove konstantne sile kugla se polako zaustavlja, opet po paraboli:

$$\triangle\vec{a_{kot}} = -\mu_k \left|\vec{F_n}\right| \hat{v} \tag{3.6}$$

gdje je μ_k konstanta trenja kotrljanja, $|\mu_k| << |\mu|$. Kugla usporava na način da održava stanje kotrljanja, stoga kutna deceleracija mora odgovarati linearnoj za zadani radijus:

$$\triangle\vec{\alpha_{kot}} = \frac{1}{R}\triangle\vec{a_{kot}} \times \hat{y} \tag{3.7}$$

Kako naš model uzima samo jednu točku kontakta, ne postoji sila koja će zaustaviti spin kugle oko y osi. Zbog toga uvodimo još jednu silu, koja djeluje u smjeru y osi, suprotno od spina kugle:

$$\triangle\alpha_y = -\mu_s sgn(\omega_y)|\vec{F_n}| \tag{3.8}$$

gdje je $sgn(x) = \frac{x}{|x|}, \quad sgn(0) = 0.$

4. Model sudara

4.1. Hertzov model kontakta

Iako kugle simuliramo krutim tijelima, ništa u prirodi nije savršeno kruto, pa tako ni plastika od koje su napravljene biljarske kugle. Kada se dva tijela sudare odnosno dodirnu, na mjestu dodira javlja se deformacija. Ovisno o sili kojom se pritišću, ta deformacija može biti veća ili manja i opire se sili koja ju je uzrokovala. Slika 4.1 pokazuje tu deformaciju za dvije kugle.

Hertzov model kontakta klasično se opisuje kao sila koja ovisi o konstanti E^*, zakrivljenosti ploha koje se sudaraju R, i dubini prodiranja δ.

$$F = \frac{4}{3} E^* R^{1/2} \delta^{3/2} \tag{4.1}$$

Zamijenimo li konstantne izraze sa C_s (nazovimo to parametar krutosti):

$$F = C_s \delta^{3/2} \tag{4.2}$$

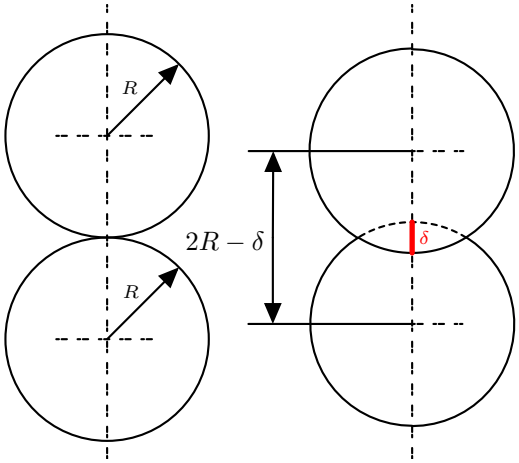
Zanemarili smo R, zakrivljenost ploha koje se sudaraju. Inače se zakrivljenosti moraju uzeti u obzir, no kako su zakrivljenosti kugli i sudarača uglavnom konstantne, možemo ih i modelirati kroz sam C_s. Jednadžba 4.2 modelira savršeno elastičan sudar pomoću nelinearne opruge. Primijetimo da ova jednadžba predstavlja diferencijalnu jednadžbu ($x < 0$ predstavlja sudar):

$$mx'' = -C_s x^{3/2} \tag{4.3}$$

Uzimamo općenitiji oblik ove jednadžbe:

$$mx'' = -C_s x^m - C_d x' \tag{4.4}$$

U ovom obliku, x je iznos prodiranja u smjeru normale, a x' je brzina u smjeru normale. Koeficijent C_s predstavlja krutost "opruge" (engl. *stiffness*). Koeficijent C_d je prigušivanje (engl. *damping*), odnosno gubitak energije. Koeficijent m



Slika 4.1: Primjer deformacije kugli pri sudaru

određuje nelinearnost opruge. Njega ćemo fiksno kodirati (engl. *hard coded*) za konkretne tipove sudarača. U slučaju kugli, ruba i površine stola $m = 3/2$.

4.2. Normalna komponenta

Konačno, za normalnu komponentu sile reakcije pišemo:

$$\vec{F_n} = \hat{n}(C_s\delta^m - C_d(\vec{v} \cdot \hat{n})) \tag{4.5}$$

gdje je $\hat{n}$ normala površine sudarača u dodirnoj točki.

4.3. Trenje

Nakon što smo izračunali normalnu komponentu sile reakcije $\vec{F_n}$, možemo definirati pravilo za silu trenja. Ovdje ćemo koristiti običnu Coulombovu silu trenja. Računamo brzinu klizanja dodirne točke:

$$\vec{u} = \vec{v} + R\hat{n} \times \vec{\omega} \tag{4.6}$$

Sila trenja djeluje u suprotnom smjeru od $\vec{u}$, iznosa $\mu|\vec{F_n}|$ u dodirnoj točki:

$$\begin{aligned} \vec{F_{tr}} &= -\mu|\vec{F_n}|\hat{u} \\ \vec{r_{tr}} &= -R\hat{n} \end{aligned} \tag{4.7}$$

4.4. Neki sudarači

Prema jednadžbama 4.5 i 4.7, sila reakcije će, između ostaloga, ovisiti o normali $\hat{n}$ i dubini prodiranja δ, koje su pak funkcije položaja kugle $\vec{p}$. To znači da će različiti sudarači imati različite funkcije $\delta(\vec{p})$ i $\hat{n}(\vec{p})$. Ostalo će biti jednako za sve sudarače, samo s drugim parametrima (C_s, C_d, μ, m).

U nastavku ćemo modelirati te funkcije za neke osnovne oblike.

4.4.1. Ravnina

Ravninu definiramo vektorom položaja $\vec{r}$ i vektorom normale $\hat{m}$. Vektor normale pokazuje u poluprostor iznad sudarača. Očigledno $\hat{n}$ ne ovisi o $\vec{p}$, odnosno $\hat{n}(\vec{p}) = \hat{m}$, a funkciju prodiranja kugle definiramo:

$$\begin{aligned} \vec{\triangle p} &= \vec{p} - \vec{r} \\ \delta(\vec{p}) &= R - \vec{\triangle p} \cdot \hat{n} \end{aligned} \tag{4.8}$$

gdje je R radijus kugle.

4.4.2. Omeđena ravnina

Prethodno definirana funkcija podijelit će čitav prostor na dva dijela. Ograničiti ravninu možemo tako da umjesto normale $\hat{m}$, definiramo dva vektora ($\vec{u}$, $\vec{v}$) koji će odrediti paralelogram nad kojim djeluje ravnina. Definiramo funkcije:

$$\begin{aligned} \hat{n}(\vec{p}) &= \hat{u} \times \hat{v} \\ \delta(\vec{p}) &= \begin{cases} R - \vec{\triangle p} \cdot \hat{n}, & 0 < \vec{\triangle p} \cdot \vec{u} < |\vec{u}|^2 \quad \wedge \\ & 0 < \vec{\triangle p} \cdot \vec{v} < |\vec{v}|^2 \\ 0, & \text{inače} \end{cases} \end{aligned} \tag{4.9}$$

4.4.3. Statična kugla

Kuglu definiramo vektorom položaja $\vec{r}$ i radijusom R_s. Funkcije normale i dubine prodiranja definiramo ovako:

$$\begin{aligned} \hat{n}(\vec{p}) &= \frac{\vec{\triangle p}}{|\vec{\triangle p}|} \\ \delta(\vec{p}) &= (R + R_s) - |\vec{\triangle p}| \end{aligned} \tag{4.10}$$

4.4.4. Kapsula

Kapsulu definiramo kao prostor koji prebriše kugla radijusa R_c dok se kreće od položaja $\vec{r_1}$ do $\vec{r_2}$. Za ovaj neobičan oblik, vrlo lako možemo definirati funkcije normale i dubine. Kapsula se zapravo sastoji od dvije kugle i valjka između njih. Treba izračunati vektor udaljenosti prema kugli 1, kugli 2 i valjku, te zatim uzeti jedan od tih vektora, ovisno gdje se kugla nalazi u odnosu na kapsulu:

$$\begin{aligned}
\vec{d_{k1}} &= \vec{p} - \vec{r_1} \\
\vec{d_{k2}} &= \vec{p} - \vec{r_2} \\
\vec{d_{valj}} &= \hat{u} \times \vec{d_{k1}} \times \hat{u}, \quad \vec{u} = \vec{r_2} - \vec{r_1} \\
\vec{\delta} &= \begin{cases} \vec{d_{k1}}, & \hat{u} \cdot \vec{d_{k1}} < 0 \\ \vec{d_{k2}}, & \hat{u} \cdot \vec{d_{k2}} > 0 \\ \vec{d_{valj}}, & \text{inače} \end{cases} \\
\hat{n}(\vec{p}) &= \frac{\vec{\delta}}{|\vec{\delta}|} \\
\delta(\vec{p}) &= (R + R_c) - |\vec{\delta}|
\end{aligned} \tag{4.11}$$

5. Implementacija

5.1. Korišteni alati

Za razvoj programske podrške ovog projekta koristio sam sljedeće alate:

Odeint. [1] C++ biblioteka za numeričko rješavanje običnih diferencijalnih jednadžbi. Razvijen je na generički način koristeći "Template Metaprograming" čime se postiže visoka fleksibilnost uz brzo izvođenje.

GLM "OpenGL Mathematics"[3] je C++ biblioteka za grafičke programe. Iz ove biblioteke koristimo klasu trodimenzionalnog vektora `glm::tvec3` i funkcije vezane za vektorski račun.

Visual Studio (2017). *Microsoft*ov IDE za pisanje izvornih kodova za razne programske jezike. Između ostalog, podržava razvoj C++ biblioteka i C# programa.

Unity3D. Višeplatformski pogon igre koji je razvio *Unity Technologies*. U ovom radu koristimo ga za prikaz rezultata u virtualnom okruženju.

5.2. C++ implementacija

5.2.1. Biblioteka Odeint

Odeint[1] je moćna biblioteka za integriranje običnih diferencijalnih jednadžbi (engl. *Ordinary differential equation, ODE*). Pogledajmo jednostavan primjer gdje ćemo ukratko pokazati kako se koristi. Prvo treba definirati tip podataka s kojim integrator radi. U ovom primjeru uzeli smo `std::vector<double>`. Zatim treba definirati funkciju sustava i funkciju koja obavještava o rezultatima izvršenih koraka.

```
typedef std::vector<double> state_type;
void harm_osc(const state_type &x,
              state_type &dxdt,
              const double t)
{
    dxdt[0] = x[1];
    dxdt[1] = -x[0] - x[1];
}
void result(const state_type &x, double t)
{
    std::cout << t << ":" << x[0] << "," << x[1] << '\n';
}
```

Funkcija sustava (`harm_osc`) prima trenutno stanje `x`, vrijeme `t` i izlazni parametar `dxdt` u kojeg zapisujemo rezultat, odnosno derivaciju stanja. Zatim, kako bismo simulirali zadani sustav, potrebno je zapisati početno stanje i pozvati funkciju `integrate`:

```
state_type x(2); // stanje je vektor s dva elementa
x[0] = 1.0; // pocni s x=1.0, p=0.0
x[1] = 0.0;
integrate(harm_osc, x, 0.0, 10.0, 0.1, result);
```

Funkcija `result` ispisivat će odrađene korake na standardni izlaz. Kao zadani integrator, funkcija `integrate` koristi `runge_kutta54_cash_karp` koračar (engl. *stepper*) s prilagodljivim korakom i kontrolom pogreške.

5.2.2. Podatkovna struktura kugle

Za opisivanje kugle koristimo strukture `BallState`, `DBallState`, `BallInfo` i klasu `Ball`.

`BallState` služi praćenju stanja kugle. Sastoji se od tri trodimenzionalna vektora: `p`, `v`, `w`. Ova struktura kad se ukalupi (engl. *cast*) u niz `double`-ova, sadrži njih 9. Dakle, naš niz kojim reprezentiramo Odeint stanje bit će tipa `std::vector<double>` duljine `length` djeljive s 9 (prva kugla prvih 9, druga kugla drugih 9, itd.). Taj niz ćemo ukalupiti u `BallState[length/9]` prilikom računanja funkcije sustava. Na taj način nam je kod uredniji i možemo koristiti vektore (`glm::dvec3`) za račun.

`DBallState` služi pojednostavljivanju zapisivanja derivacije stanja kugle. Ta-

kođer se sastoji od 3 trodimenzionalna vektora: `dp, dv, dw`. Na sličan način možemo Odeint stanje `std::vector<double>` pretvoriti u `DBallState[]`.

`BallInfo` služi slanju podataka iz C#-a C++-u (više o tome u poglavlju 5.3). Sadrži statične parametre kugle: radijus `r`, masu `m`.

`Ball` predstavlja jednu kuglu u simulaciji. Sadrži funkciju `AddRelativeForce`, koja dodaje silu `f` na relativnoj poziciji `p` u derivaciju stanja `db`. Kontstruktor klase `Ball` prima `m` i `r` te na temelju njih spremi vrijednosti m^{-1} i $I^{-1} = 5/(2mR^2)$ kako ih ne bi morali ponovno računati svaki put.

5.2.3. Klase aktora

Scena u našem simulatoru, osim kuglama, definirana je aktorima. Apstraktna klasa `Actor` predstavlja jedan takav aktor. Definira metodu `Resolve` koja prima kuglu `b`, stanje kugle `bs` i izlazni parametar, derivaciju stanja kugle `db`. U toj metodi aktor na temelju stanja kugle računa rezultantnu silu i zbroji je u derivaciju stanja kugle.

Apstraktna klasa `StaticCollider` je jedna vrsta aktora i predstavlja statičnu vrstu sudarača (engl. *Collider*). Takvi sudarači opisani su pomoću tri parametra: `C_Rest, C_Fric i C_Stif`. Redom, to su koeficijenti: C_d, μ i C_s. Konkretni razredi trebaju implementirati metodu `Calculate` koja vraća dubinu prodiranja i normalu točke sudara (spremljeno u `CollisionResult`).

Imamo tri primjera konkretnih statičnih sudarača: `PlaneCollider`, `BoundedPlaneCollider` i `CapsuleCollider`.

Kao primjer aktora koji nije sudarač, imamo `RodActor`. Ovaj aktor predstavlja štap koji drži udaljenost kugle od neke pozicije u prostoru.

5.2.4. Simulacija

Klasa `Simulation` zadužena je za definiranje same funkcije sustava i koračanje kroz simulaciju. Sadrži definiciju scene: parametri, kugle i aktori.

Ovdje su nam najzanimljivije funkcija sustava `operator()` i funkcija `Step()`:

```
1 void Simulation::operator()(const state_type &x, state_type &dxdt,
2                             const value_type t) const {
3     // ukalupljivanje vektora stanja u niz kugli
4     BallState* bs = (BallState*)&x[0];
5     DBallState* dbs = (DBallState*)&dxdt[0];
6     // za svaku kuglu...
```

```
7       for (int i = 0; i < b_count; i++) {
8           if (!balls[i].active) continue;
9           // prepisi trenutnu brzinu u dp
10          // dodaj gravitaciju u dv
11          StepBall(bs[i], dbs[i]);
12          // dodaj silu koja modelira stol
13          AddTableForce(balls[i], bs[i], dbs[i]);
14          // za svaki aktor, obradi potencijalnu akciju
15          // (npr. sudar sa sudaracem)
16          for (Actor* a : actors) {
17              a->Resolve(balls[i], bs[i], dbs[i]);
18          }
19      }
20      // za svaki par kugli...
21      for (int i = 0; i < balls.size(); i++) {
22      for (int j = i + 1; j < balls.size(); j++) {
23          if (balls[i].active || balls[i].active) {
24              BallBallCollision(balls[i], balls[j],
25                                bs[i], bs[j],
26                                dbs[i], dbs[j]);
27          }
28      }}
29  }
30
31  int Simulation::Step(value_type dt)
32  {
33      int steps_limit = 500, steps = 0;
34
35      t += dt;
36      while ((stepper.current_time() < t)) {
37          stepper.do_step(*this);
38          if (++steps >= steps_limit) {
39              // sprijeci beskonacnu petlju!
40              state = stepper.current_state();
41              goto overflowed;
42          }
43      }
44      stepper.calc_state(t, state);
```

```
45
46 overflowed:
47     // kopiraj stanje u izlazni buffer za Unity
48     for (int i = 0; i < b_count; i++) {
49     for (int j = 0; j < bsoffset; j++) {
50         ((value_type*)&(b_states[i]))[j] = state[i*bsoffset + j];
51     }}
52     return steps;
53 }
```

Ukratko ćemo još objasniti i što radi step funkcija. Glavni dio obavlja se pozivom funkcije `stepper.do_step(*this)`. Ta funkcija kaže `stepper`u koji je tipa Bulirsch-Stoer da napravi sljedeći korak, koliko god on bio velik, uz funkciju sustava `(*this)()` (odnosno `Simulation::operator()`). Dok god stepper ne preskoči željeno vrijeme, ponavljaj `do_step()`. Konačno, `stepper.calc_state(t, state)` u `state` zapiše interpolirani korak u vremenu `t`.

5.3. Unity dodatak i C# omotač

Unity3D koristi C# za definiranje ponašanja komponenti (može se koristiti i Javascript). Kako bismo povezali C# i našu C++ biblioteku, koristimo postupak koji se zove *Marshalling*. Ukratko, potrebno je uvesti eksportirane metode iz C++ biblioteke u C#, a pri pozivima prevoditi C++ strukture u C# strukture i obratno.

Pogledajmo kraći primjer. Prvo deklariramo eksportirane metode u C++ biblioteci:

```
1 #define SHARED_API __declspec(dllexport)
2 extern "C" {
3     SHARED_API Simulation* CreateSimulation(BallInfo* b_info,
4         BallState* bs, size_t b_count, SimParams simparams);
5     SHARED_API void DeleteSimulation(Simulation* s);
6     SHARED_API int StepSimulation(Simulation* s, value_type dt);
7     ...
8 }
```

Zatim na C# strani imamo sljedeći postupak uvođenja C++ metode:

```
1 delegate IntPtr _CreateSimulation([In] BallInfo[] ballinfo,
2     [In, Out] BallState[] balls, int b_count, SimParams ps);
```

```
3  delegate void _DeleteSimulation(IntPtr sim);
4  delegate int _StepSimulation(IntPtr sim, double dt);
5  // pozvati negdje u inicijalizaciji
6  IntPtr _lptr = Native.LoadLibrary(_dllpath);
7  static IntPtr CreateSimulation(BallInfo[] ballinfo,
8      BallState[] balls, SimParams ps) {
9      return Native.Invoke<IntPtr, _CreateSimulation>(_lptr,
10         ballinfo, balls, balls.Length, ps);
11 }
12
13 static void DeleteSimulation(IntPtr sim) {
14     Native.Invoke<_DeleteSimulation>(_lptr, sim);
15 }
16
17 static int StepSimulation(IntPtr sim, double dt) {
18     return Native.Invoke<int, _StepSimulation>(_lptr,
19         sim, dt);
20 }
```

Za implementaciju klase `Native` pogledati dodatak A.

Na ovaj način, razvijen je Unity dodatak (engl. *plugin*) koji sadrži C# omotače C++ struktura i klasa. Korisnik treba samo uvesti ovaj dodatak i ima sve što mu je potrebno za razvoj.

6. Rezultati

6.1. Svojstva

6.1.1. Stabilnost

Kako se ostvareni simulator oslanja na numeričku integraciju, nailazimo na problem nestabilnosti zbog skokovitih funkcija. Numeričke integracije se ne ponašaju stabilno pri skokovitim funkcijama kao što je trenje (često mijenja predznak). Iz tog razloga, skokove u predznaku treba modelirati "glatkim" funkcijama. Jedna takva glatka funkcija skoka, koju koristimo u simulatoru, je `Simulation::fricstep` (dodatak B). Uvođenjem ove funkcije skoka, unosimo malo odstupanje od teoretskog modela, ali zato dobivamo stabilnu integraciju.

6.1.2. Složenost

Vrlo je teško odrediti asimptotsku složenost adaptivnih integratora. Razlog tome je adaptivno podešavanje koraka u ovisnosti o procijenjenoj pogrešci. Ono što svakako možemo reći je da prilikom interakcija, gdje sudjeluju sile velikih iznosa (sudari), integrator će trošiti više koraka kako bi ih opisao. Okvirni broj za sudar dviju kugli je 20 do 60 koraka, ovisno o brzini kugli. Ovaj broj raste na 100 i više koraka ukoliko u sudaru sudjeluje više kugli (slučaj trokuta kugli), odaberemo li manju toleranciju pogreške itd. S druge strane, situacije konstantnih sila (kotrljanje, padanje, klizanje, dodirivanje) ne uzrokuju smanjenje koraka i izračunavaju se definiranim korakom.

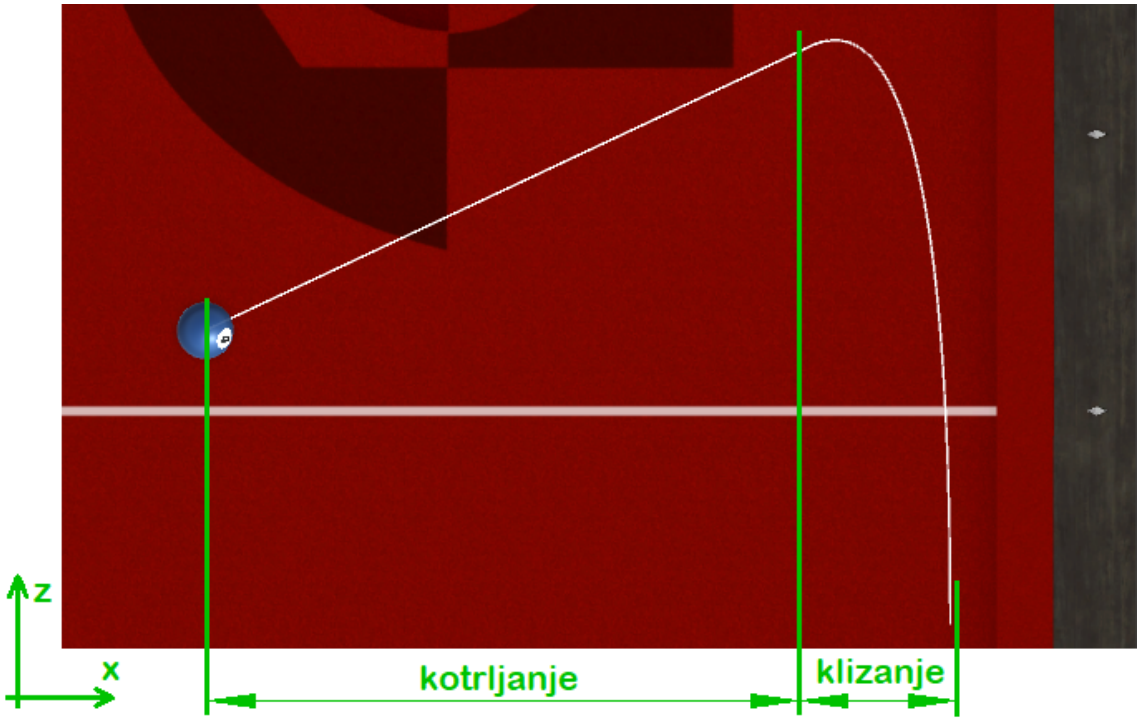
Mjesto na kojem mi možemo utjecati na brzinu jest funkcija sustava, koja se poziva u svakom od tih koraka. Kako bi provjerili sve sudare među n kugli, potrebno je raditi n^2 provjera. Dodatno, neka postoji m sudarača (odnosno aktora). To će zahtijevati mn provjera. Dakle, imamo *a priori* složenost $O(n^2 + mn)$. Ova složenost mogla bi se znatno smanjiti, npr. uporabom oktalnog stabla, međutim opis tog postupka prelazi okvire rada.

Za naše potrebe dovoljno je da se simulacija može odrađivati u stvarnom vremenu, što je većinom slučaj (testirano na procesoru Intel i5-5200U, 2.7GHz). Ponekad se dogodi da simulacija treba nešto više vremena, no to se u budućnosti može riješiti višedretvenim programiranjem, također izvan okvira rada.

6.2. Simulacije

U ovom poglavlju, simulirat ćemo neke od zanimljivih situacija ostvarenim simulatorom. Rub stola je aproksimiran kapsulama na visini 3/5 promjera kugle.

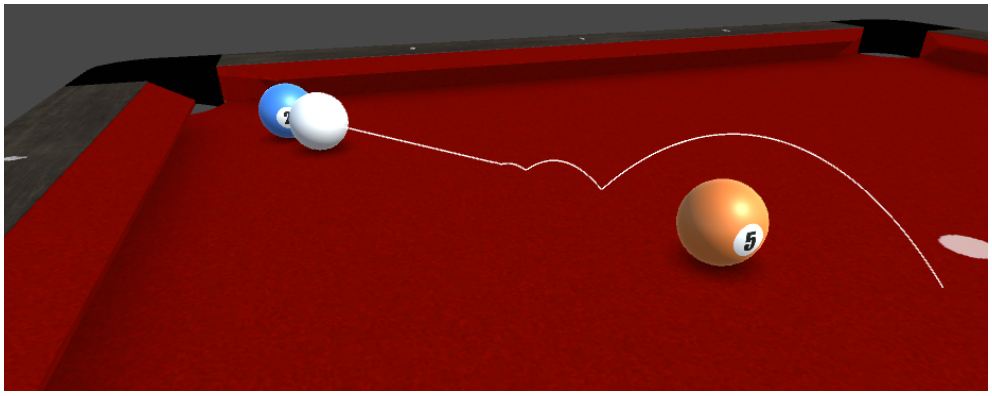
Prvi primjer (Slika 6.1) je udarac prema naprijed uz spin unazad (uz malo bočnog spina). Lopta je prvo u stanju klizanja dok se kreće prema naprijed po paraboli, zatim prelazi u stanje kotrljanja, gdje usporava po paraboli do zaustavljanja.



Slika 6.1: Osnovni primjer kretanja kugle. Klizanje (spin unazad), kotrljanje.

Na slici 6.2 vidimo mogućnost skoka kugle. Zanimljivo je vidjeti da nakon neke brzine kugla više ne odskače. To je zbog koeficijenta prigušenja C_d (više u poglavlju 4). U kontinuiranim modelima to se modelira provjerom iznosa brzine, no u iterativnom modelu simulacija sama proizvede takvo ponašanje.

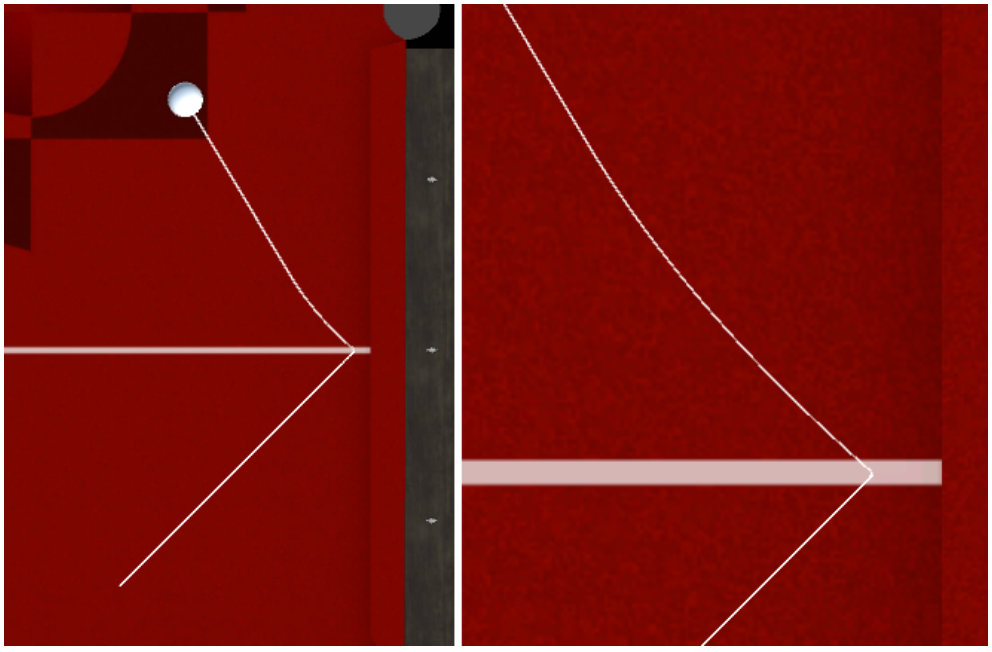
Klasičan problem odbitka kugle od ruba stola vidimo na slici 6.3. Bitno je uočiti dva učinka:



Slika 6.2: Primjer skoka kugle.

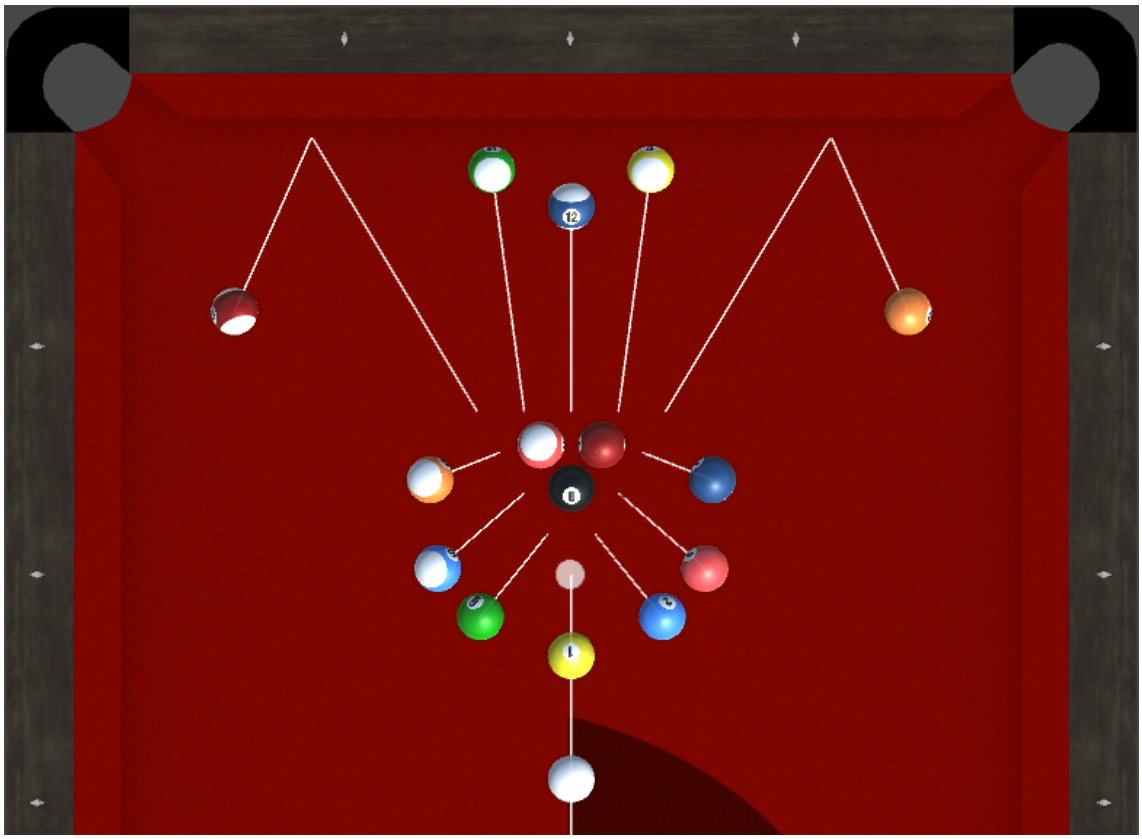
1. Kugla će kratko kliziti nakon odbitka, zbog spina kojeg dobije kotrljajući se prema rubu stola.

2. Moguće je da se kugla inicijalno odbije pod malo većim kutem nego upadnim, što je vrlo zanimljivo primjetiti, s obzirom da kugla "izgubi" količinu gibanja okomito na zid, budući da sudar nije savršeno elastičan. To je zato što sila trenja s rubom stola uzrokuje prevođenje linearnog momenta u kutni, pa kugla gubi linearni moment u smjeru ruba više nego okomito na rub.



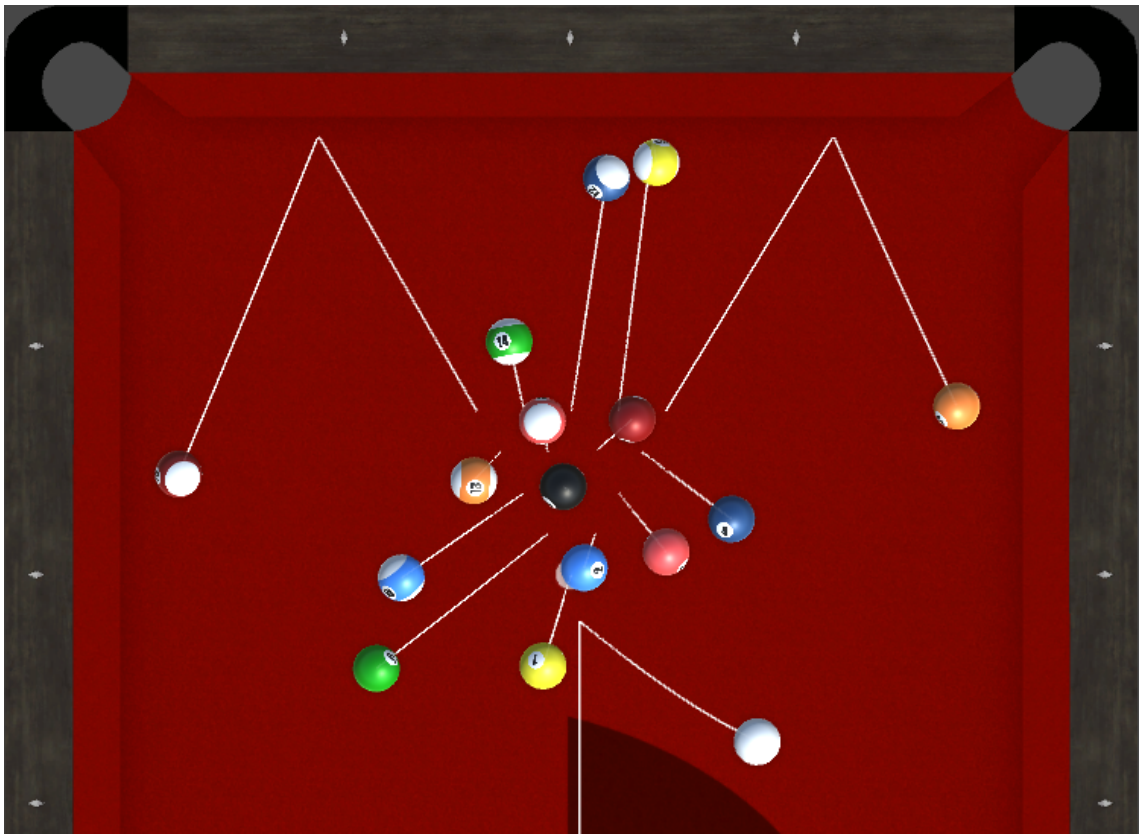
Slika 6.3: Primjer odbitka kugle od ruba stola.

Sljedeći primjer je odgovor na pitanje. Kako izgleda savršeno centralno razbijanje savršeno simetrično postavljenog trokuta kugli na biljarskom stolu? Evo odgovora (slika 6.4):



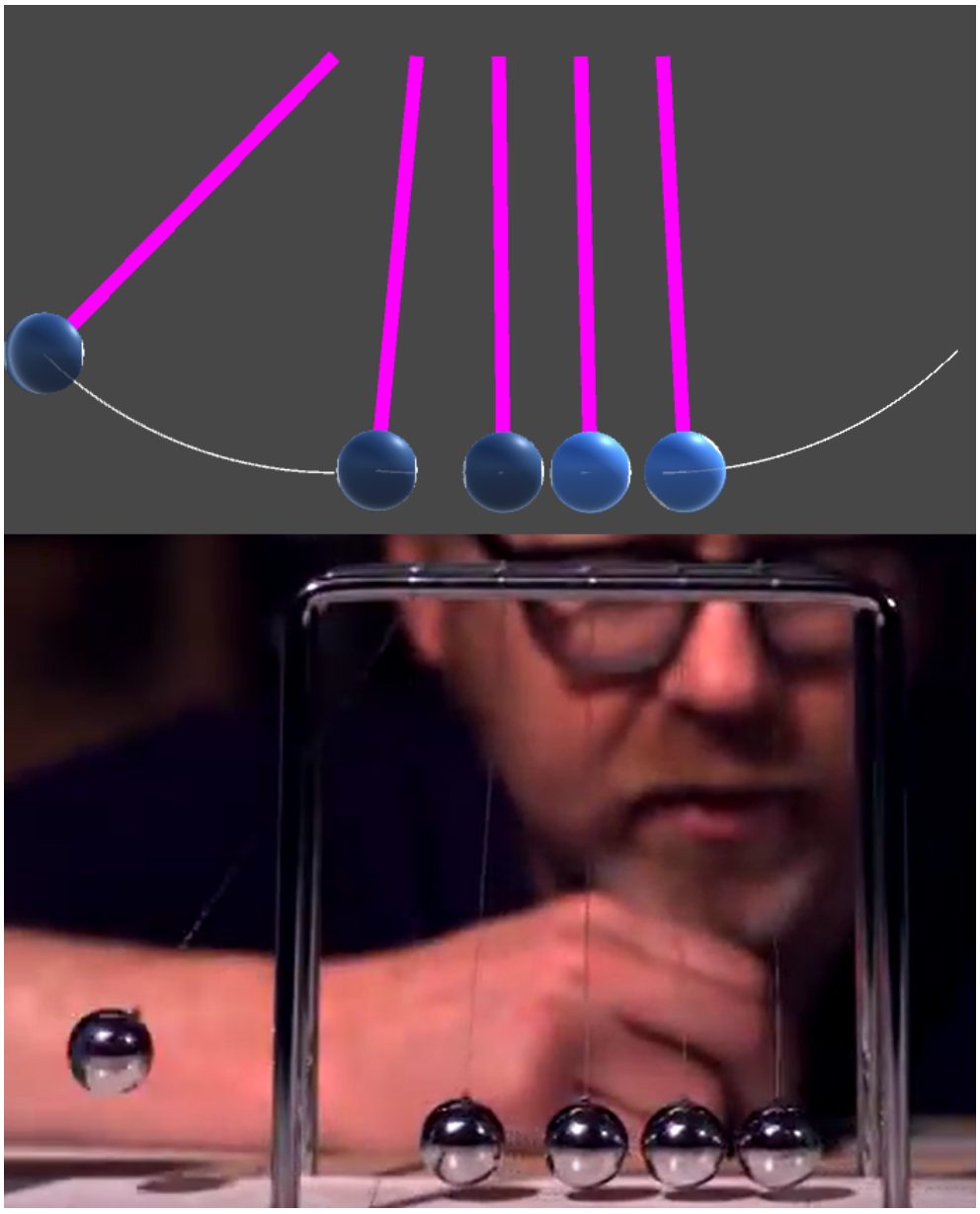
Slika 6.4: Razbijanje savršeno simetrično postavljenog trokuta.

Simulator može simulirati i "realistično" razbijanje gdje su kugle postavljene na način da svaka kugla može biti slučajno postavljena u krugu radijusa $10^{-3}R$ (R je radijus jedne kugle) u odnosu na savršenu poziciju. Bijela kugla također nije postavljena da udari savršeno centralno. Na ovom primjeru jasno vidimo osjetljivost i kompleksnost samog sustava. Baš iz ovog razloga svi simulatori biljara mogu biti samo grube aproksimacije (teškoće pri definiranju zakona sudara [2]).



Slika 6.5: Razbijanje realistično postavljenog trokuta (uz slučajni razmak).

Zadnji primjer (slika 6.6) nije vezan za biljar. Radi se o Newtonovom njihalu: n kuglica poredanih jedna iza druge koje vise na koncima zanemarive mase i konačne duljine. Konac u ovom slučaju simuliramo aktorom `Rod` koji drži razmak između kugle i neke točke u prostoru. Ako jednu kuglu odmaknemo i pustimo je da se sudari, očekujemo da će jedna kugla s druge strane odletiti, a ostale ostati stajati. Međutim to nije potpuno istina. Stvarne kugle će se odbiti kako je prikazano na slici. Zadnja kugla odleti najdalje, predzadnja blago za njom, a ostale nazad (prva kugla malo više):



Slika 6.6: Newtonovo njihalo. Gore simulacija, dolje isječak YouTube videa https://www.youtube.com/watch?v=BiLq5Gnpo8Q

7. Zaključak

U ovom radu usporedili smo dva načina simulacije kugli: kontinuirani i numerički. Nakon toga smo detaljnije proučili kako modelirati diferencijalnu jednadžbu za numeričko rješavanje. Pokazali smo kako modelirati općeniti sudar i neke osnovne sudarače. Zatim smo načinili C++ biblioteku, koju smo pomoću marshallinga uveli u C#, za korištenje kao dodatak Unityu. Konačno, pokazali smo neke zanimljive primjere koje je moguće simulirati u ostvarenom simulatoru.

Numerička integracija popularan je alat, kako u video igrama tako i u ozbiljnijim primjenama. Posebice ako je potrebno modelirati kompleksne sustave s mnoštvom materijalnih točaka i njihovih interakcija. Često je daleko jednostavnije definirati sile nego komplicirana pravila interakcije, pa ovakvi sustavi bolje opisuju stvarnost.

Uz malo optimizacije i dorade programske potpore, ovaj simulator mogao bi se koristiti za razne svrhe. Između ostalog i kao odličan pogon neke igre biljara, stolnog tenisa (bilo bi potrebno dodati Magnusov učinak kao aktora), boćanja (modelirati neravan teren) i sl. "Običnom" igraču biljara možda je dovoljan kontinuirani pristup, no profesionalcima će i najmanji nedostatak realističnosti zasmetati. Osim toga, ako imamo realističan model, ljudima će biti zanimljivije igrati, jer će biti potrebno mnogo više vježbe kako bi savladali tehniku igranja.

Literatura

[1] Karsten Ahnert i Mario Mulansky. Odeint. http://headmyshoulder.github.io/odeint-v2/doc/index.html, 2009-2012.

[2] A. Chatterjee. *Rigid body collisions: some general considerations, new collision laws, and some experimental data.* Cornell University, January, 1997.

[3] Khronos™ group. Opengl mathematics (glm). http://glm.g-truc.net/0.9.8/index.html, 2005-2017.

[4] Yan-Bin Jia, Matthew T Mason, i Michael A Erdmann. Trajectory of a billiard ball and recovery of its initial velocities. 2011.

Dodatak A
Klasa Native

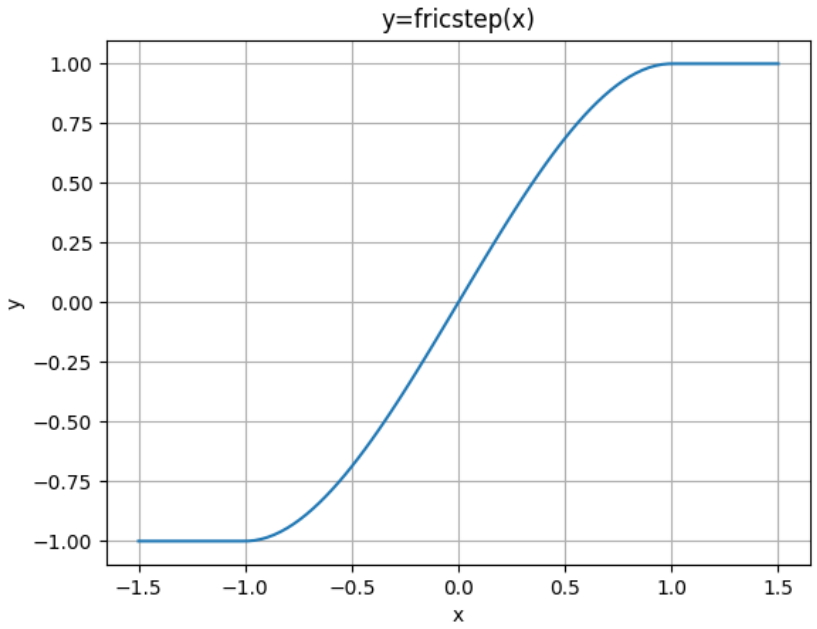
```
1  /*
2  * Native dll invocation helper by Francis R. Griffiths-Keam
3  * www.runningdimensions.com/blog
4  */
5
6  using System;
7  using System.Runtime.InteropServices;
8  using UnityEngine;
9
10 public static class Native
11 {
12     public static T Invoke<T, T2>(IntPtr library, params object[] pars)
13     {
14         string name = typeof(T2).Name.TrimStart('_');
15         IntPtr funcPtr = GetProcAddress(library, name);
16         if (funcPtr == IntPtr.Zero)
17         {
18             Debug.LogWarning("Could not gain reference to method address.
                   " + name);
19             return default(T);
20         }
21
22         var func = Marshal.GetDelegateForFunctionPointer(funcPtr,
               typeof(T2));
23         return (T)func.DynamicInvoke(pars);
24     }
25
```

```
26      public static void Invoke<T>(IntPtr library, params object[] pars)
27      {
28          string name = typeof(T).Name.TrimStart('_');
29          IntPtr funcPtr = GetProcAddress(library, name);
30          if (funcPtr == IntPtr.Zero)
31          {
32              Debug.LogWarning("Could not gain reference to method address.
                    " + name);
33              return;
34          }
35
36          var func = Marshal.GetDelegateForFunctionPointer(funcPtr,
                typeof(T));
37          func.DynamicInvoke(pars);
38      }
39
40      [DllImport("kernel32", SetLastError = true)]
41      [return: MarshalAs(UnmanagedType.Bool)]
42      public static extern bool FreeLibrary(IntPtr hModule);
43
44      [DllImport("kernel32", SetLastError = true, CharSet = CharSet.Unicode)]
45      public static extern IntPtr LoadLibrary(string lpFileName);
46
47      [DllImport("kernel32")]
48      public static extern IntPtr GetProcAddress(IntPtr hModule, string
            procedureName);
49  }
```

Dodatak B
Funkcija skoka fricstep

Funkcija `fricstep` koristi se za zaglađivanje skokova pri promjeni predznaka sile trenja. U nju obično ubacimo skalar brzine pomnoženu s nekim velikim brojem, tako da su granice skoka puno bliže nuli. Graf same funkcije vidimo na slici B.1.

```
1 double fricstep(double fric) {
2     fric = clamp(fric, -1, 1);
3     fric *= 0.5;
4     fric += 0.5;
5     return fric * fric * (3 - 2 * fric) * 2 - 1;
6 }
```



Slika B.1: Funkcija zaglađivanja, `fricstep`

Fizikalno temeljena simulacija sudaranja kugli

Sažetak

Ovaj rad predstavlja fizikalno temeljenu simulaciju kretanja kugli, njihovog međusobnog sudaranja i sudaranja kugli sa statičnim predmetima. Glavni cilj je definirati općeniti matematički model, a zatim ga upotrijebiti za kreiranje sofisticiranog simulatora igre biljara. Ostvareni simulator uspješno opisuje razne zanimljive učinke koji se javljaju tijekom igre.

Ključne riječi: biljar, numerička integracija, fizika, kugla, ODJ

Physically Based Ball Collision

Abstract

This report presents a physics based method for simulating motion of spherical rigid bodies, their collisions and collisions with a variety of static colliders. The main idea is to define a general mathematical model and use it to create a sophisticated billiard game simulator. The realized simulator is successful at expressing many obscure effects that arise in a game of pool.

Keywords: billiards, pool, numerical integration, sphere, physics, ODE