SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 223

SIMULACIJA LOMLJENJA ČVRSTIH TIJELA

Alen Kralj

Zagreb, lipanj 2008.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 223

SIMULACIJA LOMLJENJA ČVRSTIH TIJELA

Alen Kralj

Zagreb, lipanj 2008.

Na ovu stranicu stavite izvornik zadatka završnog rada.

Sadržaj

Uvod

U ovom radu bit će obrađen problem simulacije lomljenja čvrstih tijela. Kako bi shvatili osnovne koncepte sustava za simulaciju fizike u poglavlju 1 prezentiran je osnovni fizikalni model kretanja čvrstih tijela. Nakon toga razrađen je implementacijski dio, te u poglavlju 2 načinjen pregled tri gotova sustava za simulaciju fizike: SPE (2.1), ODE (2.2) i BPL (2.3). Navedene su specifičnosti svake jezgre sustava i predstavljena moguća rješenja aplikacije lomljenja čvrstih tijela na svakom sustavu.

Uz ovaj rad priložen je medij s trenutnim verzijama spomenutih sustava za simulaciju fizike, DirectX SDK 2007, aplikacije lomljena čvrstih tijela i njihovi pripadajući izvorni kôdovi. Dodatak A: Dokumentacija aplikacije simulacije lomljenja čvrstih tijela detaljnije opisuje glavnu aplikaciju lomljenja čvrstih tijela priloženu uz ovaj rad.

1. Osnove modeliranja simulacije fizike čvrstog tijela

Ovo poglavlje pokriva osnove modeliranja simulacije fizike čvrstog tijela. Poglavlje je podijeljeno u dva dijela: 1.1 i 1.2. U poglavlju 1.1 govorit ćemo o simulaciji jednog čvrstog tijela: njegovo kretanje, ponašanje pod utjecajem vanjske sile i drugo, dok poglavlje 1.2 pokriva simulaciju više čvrstih tijela, odnosno njihovo međudjelovanje.

1.1. Simulacija jednog čvrstog tijela

1.1.1. Vektor stanja

Simulacija kretanja čvrstog tijela je slična simulaciji kretanja čestice. Način na koji simuliramo čestični sustav je da imamo funkciju $x(t)$ koja opisuje lokaciju čestice u svijetu u trenutku t, te funkciju $v(t) = \frac{d}{dt}x(t)$ koja nam daje brzinu čestice u trenutku t. Stoga za jednu česticu imamo vektor stanja $\boldsymbol{X}(t)$, koji je jednak $\boldsymbol{X}(t) = \begin{pmatrix} x(t) \\ v(t) \end{pmatrix}$. Kada to generaliziramo na sustav sa n čestica vektor stanja glasi:

$$\boldsymbol{X}(t) = \begin{pmatrix} x_1(t) \\ v_1(t) \\ \\ \vdots \\ \\ x_n(t) \\ v_n(t) \end{pmatrix}.$$

Da bi mogli simulirati kretanje čestice potrebna nam je još jedna informacija: sila na česticu u trenutku t. Definirat ćemo funkciju $F(t)$ kao sumu svih sila koje djeluju na česticu (gravitacija, vjetar i drugo). Ako je m masa čestice, onda promjena vektora stanja $\boldsymbol{X}$ možemo izrazit kao:

$$\frac{d}{dt}\boldsymbol{X}(t) = \frac{d}{dt}\begin{pmatrix} x(t) \\ v(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ F(t)/m \end{pmatrix}.$$

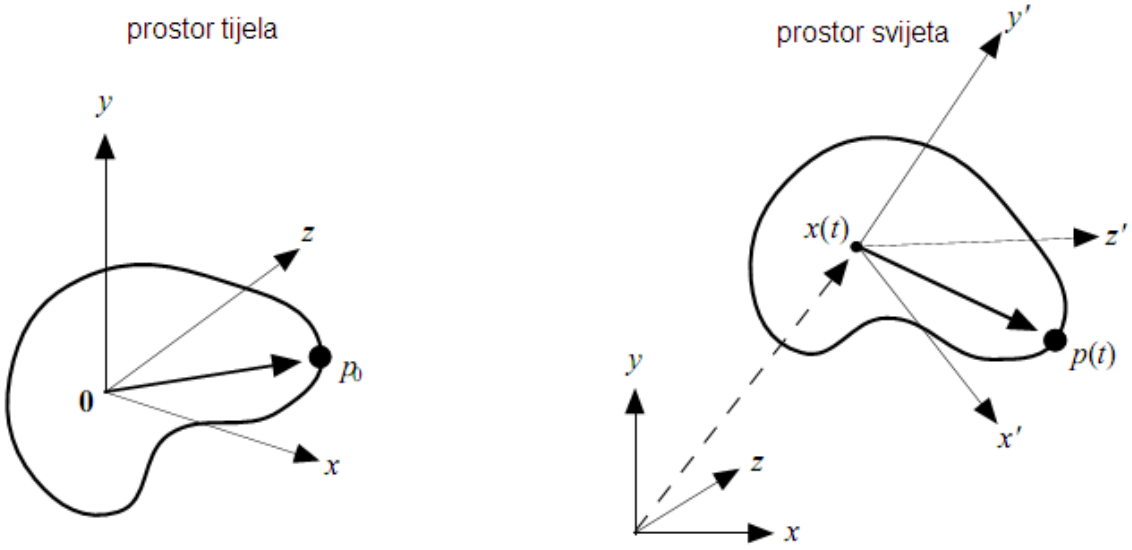
Vektor stanja kretanja čvrstog tijela je sličan gore spomenutim vektorima stanja čestica, samo što je malo kompleksniji, odnosno sadrži više informacija.

1.1.2. Pozicija i orijentacija

Poziciju čestice u svijetu u trenutku t opisujemo sa vektorom $x(t)$, koji opisuje translaciju čestice od ishodišta. Čvrsta tijela su kompliciranija, jer osim translacije, mogu biti i rotirana. Stoga za lokaciju čvrstog tijela u svijetu koristit ćemo vektor $x(t)$, koji opisuje translaciju tijela. Rotaciju ćemo s druge strane opisivati sa 3 x 3 matricom rotacije $R(t)$. Vektor $x(t)$ i matrica $R(t)$ nazivamo prostornim varijablama čvrstog tijela.

Čvrsto tijelo, za razliku od čestice, zauzima volumen u prostoru i ima karakterističnu oblik. Zato što čvrsto tijelo podliježe samo translaciji i rotaciji, odnosno fiksno je i ne mijenja oblik, definiramo prostor koji zovemo prostor tijela. Pomoću geometrijskih opisa tijela u prostoru tijela, koristimo $x(t)$ i $R(t)$ kako bi transformirali čvrsto tijelo iz prostora tijela u prostor svijeta. Slika 1.1 prikazuje tu transformaciju. Da bi pojednostavili jednadžbe, centar mase tijela smo postavili u ishodište prostora tijela (0, 0, 0). Ukratko, centar mase je točka čvrstog tijela koja pokazuje na geometrijski centar tijela. Mi zahtijevamo da geometrijski centar leži u točki ishodišta u prostoru tijela, jer ako nam je $R(t)$ rotacija tijela oko centra mase, onda pomoću fiksnog vektor $\vec{r}$ u prostoru tijela možemo rotirati tijelo i u prostoru svijeta koje je jednako izrazu $R(t)\vec{r}$. Sukladno tome ako je p_0 točka koju promatramo na čvrstom tijelu u prostoru tijela, tada u prostoru svijeta lokacija $p(t)$ točke p_0 je rezultat prvo rotacije p_0 oko ishodišta, a zatim translacije. Jednadžba koja ovo opisuje slijedi u nastavku:

$$p(t) = R(t)p_0 + x(t) .$$



Slika 1.1 Transformacija čvrstog tijela iz prostora tijela u prostor svijeta $x' = R(t)x$, $y' = R(t)y$, $z' = R(t)z$

Matrica rotacije glasi:

$$R(t) = \begin{pmatrix} r_{xx} & r_{yx} & r_{zx} \\ r_{xy} & r_{yy} & r_{zy} \\ r_{xz} & r_{yz} & r_{zz} \end{pmatrix} ,$$

gdje svaki stupac opisuje usmjerenje osi: x, y i z.

1.1.3. Brzina kretanja

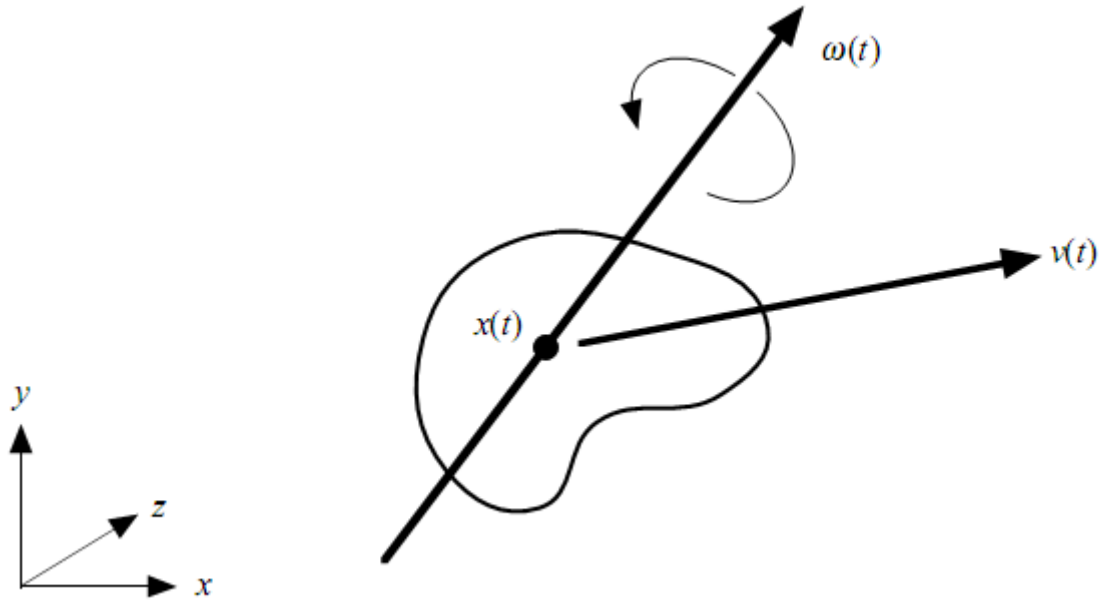
Zbog jednostavnosti $x(t)$ i $R(t)$ nazivamo *pozicijom* i *orijentacijom* tijela u trenutku t. Slijedeća stvar koju trebamo definirati je kako se pozicija i orijentacija mijenja kroz vrijeme. Točnije trebamo izraziti jednadžbe sa $\dot{x}(t)$ i $\dot{R}(t)$. Pošto je $x(t)$ pozicija centra mase u prostoru svijeta, $\dot{x}(t)$ je brzina centra mase u prostoru svijeta. Definirat ćemo linearnu brzinu kao brzinu kretanja tijela:

$$v(t) = \dot{x}(t) .$$

Ako zamislimo da je orijentacija tijela fiksna, onda tijelo može podlijeći samo translaciji. Vrijednost $v(t)$ daje brzinu te translacije.

1.1.4. Rotacija tijela oko vlastite osi

Kao dodatak translaciji, čvrstom tijelu dodajemo mogućnost rotacije oko vlastite osi, takozvani *spin.* Zamislimo da zamrznemo poziciju centra mase u prostoru. Svako pomicanje točke tijela mora biti oko osi koja prolazi kroz centar mase. U suprotnom bi se centar mase pomicao. Tu rotaciju opisujemo pomoću kutne brzine $\omega(t)$. Smjer kutne brzine $\omega(t)$ nam daje smjer osi oko koje se tijelo okreće. Slika 1.2 prikazuje razliku između linearne brzine $v(t)$ i kutne brzine $\omega(t)$.

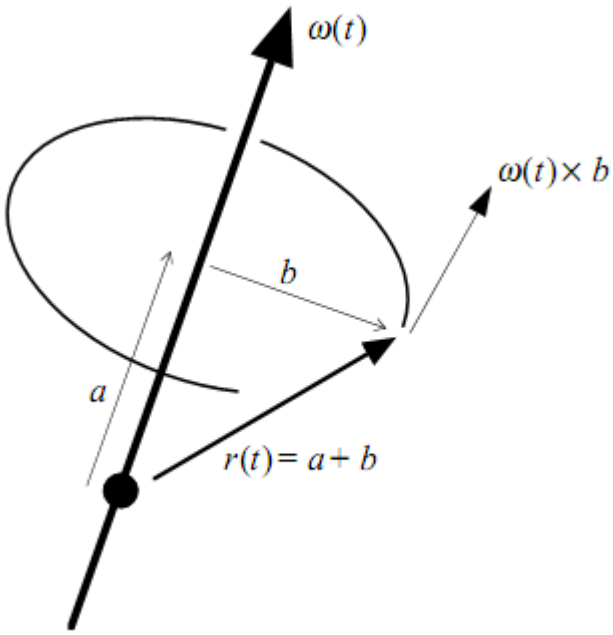


Slika 1.2 Linearna brzina $v(t)$ i kutna brzina $\omega(t)$

Apsolutna vrijednost od $\omega(t)$, odnosno $|\omega(t)|$ govori koja je brzina rotacije. Točnije opisuje koji kut će tijelo prevaliti u vremenu t, ako naravno kutna brzina ostane konstantna.

Kod linearne brzine $x(t)$ i $v(t)$ smo povezali pomoću $v(t) = \dot{x}(t)$. Zanima nas kako su $R(t)$ i $\omega(t)$ povezani? Očigledno da $\omega(t) = \dot{R}(t)$ ne može vrijediti, jer je $R(t)$ matrica, a $\omega(t)$ je vektor. Da bi dobili njihovu povezanost moramo se podsjetiti što nam govori $R(t)$ matrica. U poglavlju 1.1.2 spomenuli smo da svaki stupac unutar $R(t)$ matrice opisuje smjer svake osi (x, y, z). Dakle onda $\dot{R}(t)$ bi govorila brzinu kojom se osi transformiraju. Slika 1.3 prikazuje tijelo sa kutnom brzinom $\omega(t)$. Zamislimo vektor $r(t)$ u vremenu t u

prostoru svijeta koji je fiksni na tijelu, odnosno vektor $r(t)$ se kreće zajedno sa čvrstim tijelom kroz prostor svijeta. Pošto je $r(t)$ vektor smjera, translacija ne utječe na njega. To znači da je $\dot{r}(t)$ neovisan o $v(t)$. Da bi proučili vektor $\dot{r}(t)$, rastavit ćemo vektor $r(t)$ na vektor a i vektor b, gdje je vektor a paralelan sa $\omega(t)$, a vektor b okomit na $\omega(t)$. Ako je kutna brzina konstantna, vrh vektora $r(t)$ tvori krug oko $\omega(t)$, čiji je radijus $|b|$. Iz toga slijedi da je brzina $r(t)$ jednaka $|b||\omega(t)|$.



Slika 1.3 Tijelo sa kutnom brzinom $\omega(t)$

Pošto su b i $\omega(t)$ međusobno okomiti vrijedi:

$$|b \times \omega(t)| = |b||\omega(t)|.$$

Kada uzmemo sve spomenuto, dobivamo:

$$\dot{r}(t) = \omega(t) \times b.$$

Ovo možemo još razraditi pošto $r(t) = a + b$, te pošto je a paralelan sa $\omega(t)$, njihov umnožak je jednak nuli. Prema tome dobivamo:

$$\dot{r}(t) = \omega(t) \times b = \omega(t) \times b + \ \omega(t) \times a = \ \omega(t) \times (b + a)$$

Te konačno dobivamo:

$$\dot{r}(t) = \omega(t) \times r(t).$$

Isto vrijedi i za matrice:

$$\dot{R}(t) = \omega(t)^* R(t),$$

gdje je $\omega(t)^*$ matrica na temelju vektora $\omega(t)$, koja glasi:

$$\omega(t)^* = \begin{pmatrix} 0 & -\omega(t)_z & \omega(t)_y \\ \omega(t)_z & 0 & -\omega(t)_x \\ -\omega(t)_y & \omega(t)_x & 0 \end{pmatrix}.$$

1.1.5. Masa tijela

Da bi pojednostavili derivacije, te integracije, uzet ćemo da se tijelo sastoji od većeg broja manjih dijelova (čestica). Te čestice obično indeksiramo od 1 do N. Svaka čestica (i), ima masu m_i, te konstantan vektor r_{oi} u prostoru tijela. Prema tome lokacija i-te čestice u prostoru svijeta u vrijeme t iznosi:

$$r_i(t) = R(t) r_{0i} + x(t).$$

Ukupna masa tijela iznosi:

$$M = \sum_{i=1}^{N} m_i.$$

1.1.6. Brzina čestice

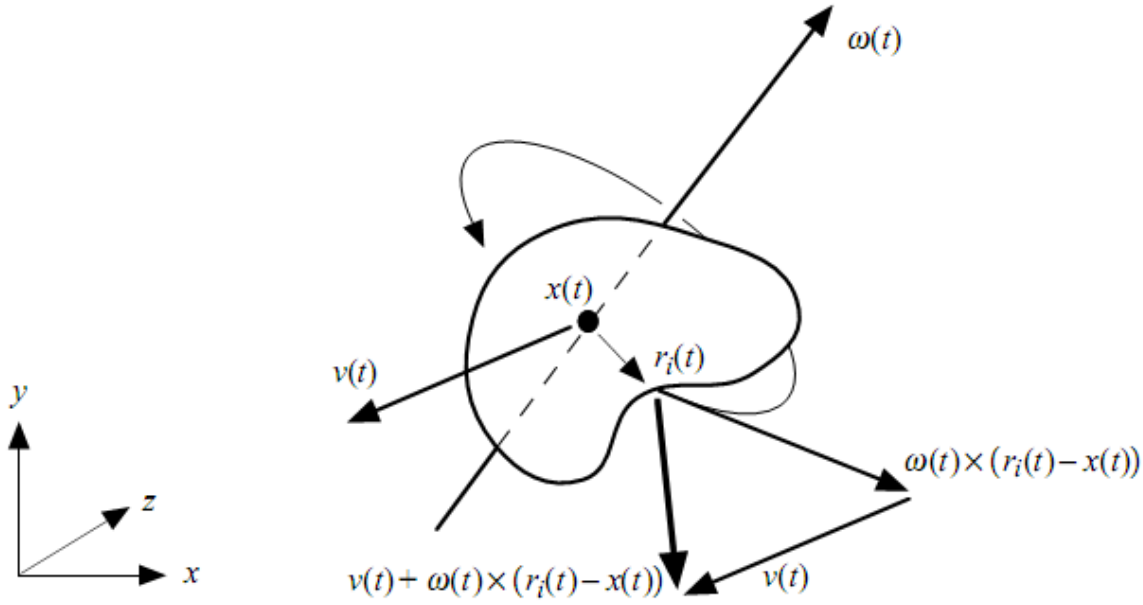
Brzinu $\dot{r}_i(t)$ čestice i dobivamo pomoću već spomenute relacije $\dot{R}(t) = \omega(t)^* R(t)$, tako da ju zapišemo vektorski pomoću:

$$\dot{r}_i(t) = \omega(t)^* R(t) r_{0i} + v(t).$$

Navedeno možemo raspisati na slijedeći način:

$$\dot{r}_i(t) = \omega(t)^* R(t) r_{0i} + v(t)$$
$$\dot{r}_i(t) = \omega(t)^* \big(R(t) r_{0i} + x(t) - x(t)\big) + v(t)$$
$$\dot{r}_i(t) = \omega(t)^* \big(r_i(t) - x(t)\big) + v(t)$$
$$\dot{r}_i(t) = \omega(t) \times \big(r_i(t) - x(t)\big) + v(t)$$

Slika 1.4 prikazuje brzinu čestice u prostoru svijeta. Brzinu čestice možemo rastaviti na vektor linearne brzine $v(t)$ i vektor kutne brzine $\omega(t) \times \big(r_i(t) - x(t)\big)$.



Slika 1.4 Brzina i-te čestice u prostoru svijeta

1.1.7. Centar mase

Centar mase tijela u prostoru svijeta je definiran kao:

$$\frac{\sum m_i r_i(t)}{M},$$

gdje je M masa tijela, odnosno zbroj svih masa m_i. Kada govorimo da koristimo koordinatni sustav centra mase, znači da u prostoru tijela vrijedi:

$$\frac{\sum m_i r_i(t)}{M} = \mathbf{0}.$$

Primijetimo da vrijedi i $\sum m_i r_i(t) = \mathbf{0}$. Slijedeći izraz dokazuje da je $x(t)$ lokacije centra mase u vremenu t:

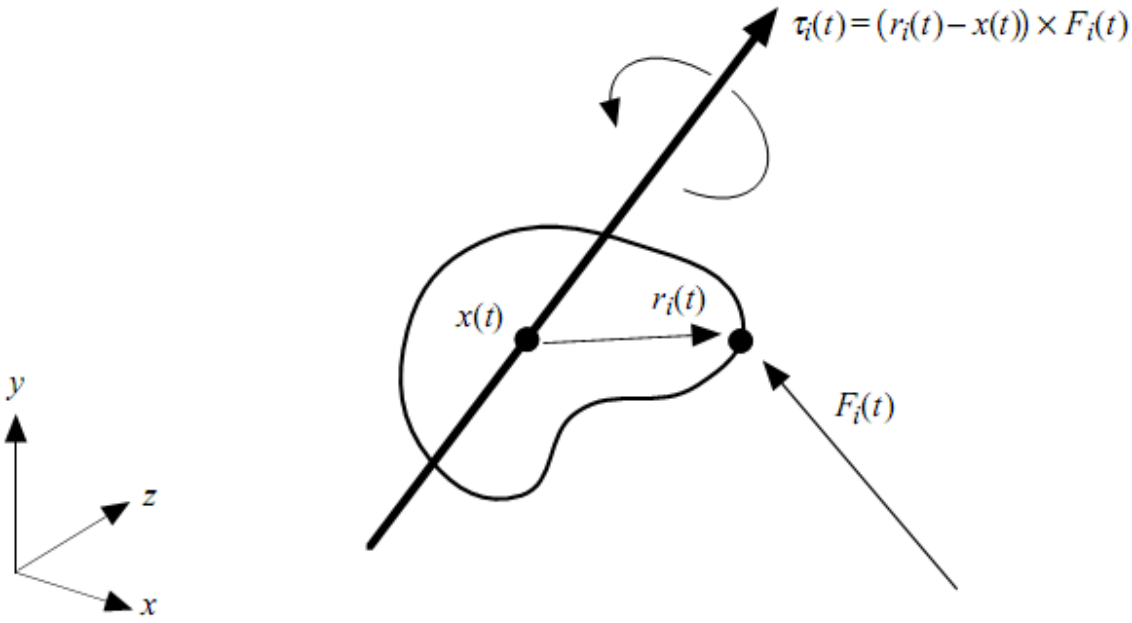
$$\frac{\sum m_i r_i(t)}{M} = \frac{\sum m_i (R(t) r_{0i} + x(t))}{M} = \frac{R(t) \sum m_i r_{0i} + \sum m_i \, x(t)}{M} = x(t) \frac{\sum m_i}{M} = x(t).$$

1.1.8. Sila i moment sile

Kada zamišljamo silu koja djeluje na tijelo zbog nekog vanjskog utjecaja, na primjer gravitacija ili vjetar, zapravo zamišljamo silu koja djeluje na određenu česticu tijela. Definiramo silu $F_i(t)$ kao ukupan zbroj vanjskih sila koje djeluju na česticu i u vremenu t. Isto tako definiramo vanjski moment sile $\tau_i(t)$ koji djeluje na česticu i koji glasi:

$$\tau_i(t) = (r_i(t) - x(t)) \times F_i(t)$$

Razlika između momenta sile i sile je što moment sile ovisi o lokaciji $r_i(t)$ čestice, odnosno udaljenost od centra mase $x(t)$. Intuitivno možemo zamisliti smjer momenta sile $\tau_i(t)$ kao os oko koje će se tijelo okretati zbog utjecaja sile $F_i(t)$. Slika 1.5 prikazuje relaciju momenta sile, sile i udaljenost od centra mase.



Slika 1.5 Moment sile $\tau_i(t)$ koji nastaje zbog utjecaja sile $F_i(t)$

Ukupna sila $F(t)$ koja djeluje na tijelo je jednaka:

$$F(t) = \sum F_i(t),$$

dok je ukupan moment sile jednak:

$$\tau(t) = \sum \tau_i(t) = \sum (r_i(t) - x(t)) \times F_i(t).$$

Primijetimo dakle da $F(t)$ ne govori kako sila djeluje na tijelo, dok $\tau(t)$ govori kako je sila distribuirana po tijelu.

1.1.9. Linearna količina gibanja

Linearna količina gibanja p čestice sa masom m i brzinom v je definirana kao:

$$p = mv.$$

Ukupna linearna količina gibanja $P(t)$ čvrstog tijela je onda jednaka:

$$P(t) = \sum m_i \dot{r}_i(t).$$

Koristeći već spomenutu jednakost $\dot{r}_i(t) = \omega(t) \times \big(r_i(t) - x(t)\big) + v(t)$ ukupnu količinu gibanja možemo raspisati kao:

$$P(t) = \sum m_i \dot{r}_i(t)$$

$$P(t) = \sum (m_i \omega(t) \times \big(r_i(t) - x(t)\big) + m_i v(t))$$

$$P(t) = \omega(t) \times \sum m_i \big(r_i(t) - x(t)\big) + \sum m_i v(t).$$

Pošto koristimo koordinatni sustav sa središtem centra mase lijevi dio jednadžbe je jednak nuli. Prema tome dobivamo:

$$P(t) = \sum m_i v(t)$$

$$P(t) = \Big(\sum m_i\Big) v(t)$$

$$P(t) = Mv(t).$$

Rezultat je jako lijepa relacija koja ukupnu linearnu količinu gibanja čvrstog tijela opisuje kao da je tijelo čestica mase M i brzine $v(t)$. Zbog toga imamo jednostavne transformacije između $P(t)$ i $v(t)$: $P(t) = Mv(t)$ i $v(t) = P(t)/M$, te pošto je M konstanta:

$$\dot{v}(t) = \frac{\dot{P}(t)}{M}.$$

1.1.10. Kutna količina gibanja

Koncept kutne količine gibanja uvodimo kako bi pojednostavili jednadžbe. Uvodimo definiciju ukupne kutne količine gibanja $L(t)$, koja je jednaka:

$$L(t) = I(t)\omega(t),$$

gdje je $I(t)$ matrica 3 x 3. $I(t)$ nazivamo tenzorom momenta tromosti, odnosno tenzor ranga 2, te će biti detaljnije obrađen u sljedećem poglavlju. Veza između kutne količine gibanja i momenta sile definirana je kao relacija:

$$\dot{L}(t) = \tau(t).$$

1.1.11. Tenzor momenta tromosti

Tenzor momenta tromosti nam govori kako je masa raspodijeljena po tijelu s obzirom na centar mase. U matematici uvodimo tenzor kao skup elemenata određenog ranga (reda), te kažemo da se sastoji od ukupno 3^{rang} komponenata. Tenzor momenta tromosti je ranga 2, te se sastoji od 9 komponenta:

$$I(t) = \begin{pmatrix} I_{xx}(t) & I_{xy}(t) & I_{xz}(t) \\ I_{yx}(t) & I_{yy}(t) & I_{yz}(t) \\ I_{zx}(t) & I_{zy}(t) & I_{zz}(t) \end{pmatrix}.$$

Tenzor je simetričan, odnosno $I_{ij} = I_{ji}$. Slijedeća relacija nam je od znatne važnosti:

$$I(t) = R(t) I_{body} R(t)^T,$$

gdje je matrica I_{body} jednaka:

$$I(t) = \sum m_i ((r_{0i}{}^T r_{0i})\mathbf{1} - r_{0i} r_{0i}{}^T).$$

1.1.12. Jednadžba gibanja čvrstog tijela

Sada konačno kada smo pokrili sve koncepte vektora stanja čvrstog tijela možemo napisati jednadžbu gibanja. Dakle vektor stanja čvrstog tijela jednak je:

$$\boldsymbol{X}(t) = \begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix},$$

gdje je $x(t)$ pozicija, $R(t)$ matrica rotacije, $P(t)$ linearna količina gibanja, te $L(t)$ kutna količina gibanja čvrstog tijela. Kada deriviramo po vremenu vektor stanja dobivamo željenu jednadžbu:

$$\frac{d}{dt}\boldsymbol{X}(t) = \frac{d}{dt}\begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ \omega(t) * R(t) \\ F(t) \\ \tau(t) \end{pmatrix},$$

gdje je $v(t)$ linearna brzina, $\omega(t) * R(t)$ kutna brzina, $F(t)$ sila, te $\tau(t)$ ukupan moment sile čvrstog tijela. Više informacija može se naći u [1, 3, 4].

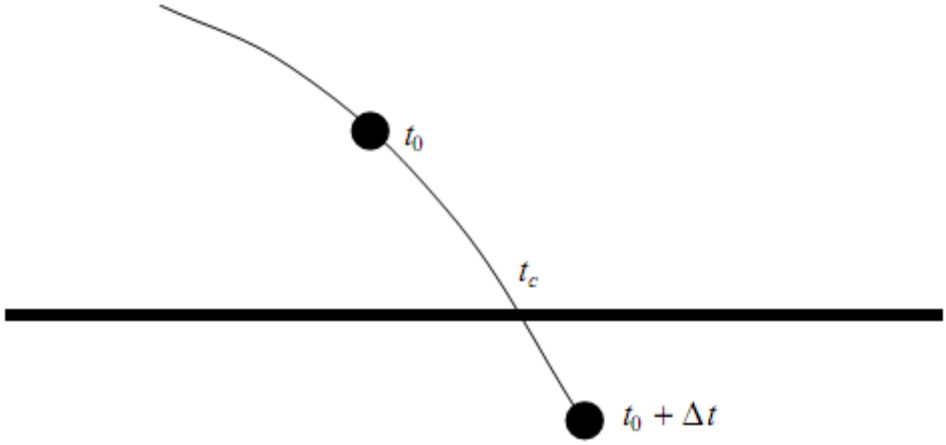
1.2. Simulacija više čvrstih tijela

Sada kada znamo modelirati jedno čvrsto tijelo, vrijeme je da pređemo na nešto kompliciraniji model, odnosno da uvedemo više čvrstih tijela, te opišemo njihovo međudjelovanje. Pošto govorimo o čvrstim tijelima, prilikom sudara dvaju takvih tijela ne dopuštamo ni najmanju penetraciju jednog tijela unutar drugog. Ovaj problem nazivamo

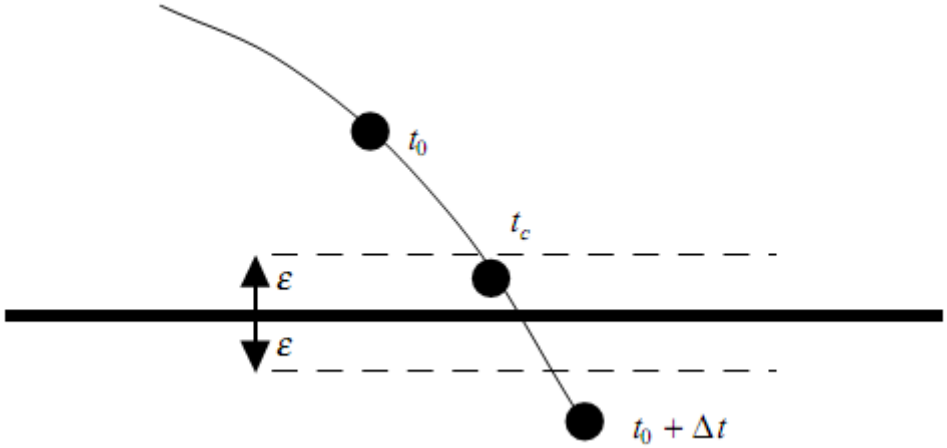
problem ne-penetrirajućih tijela. Zašto je ovo uopće problem, te gdje se krije izvor problema, a gdje rješenje slijedi u nastavku. U [2, 5, 6, 7, 8] se može naći više informacija.

1.2.1. Problem ne-penetrirajućih tijela

U prethodnom poglavlju definirali smo gibanje jednog čvrstog tijela. Navedeni model slijedi opće poznate zakone mehanike (fizike) tijela, te zahtjeva deriviranje po vremenu u jednadžbi kretanja. Pošto jednadžba zahtjeva realnu vremensku domenu, a mi želimo simulirati na računalu u diskretnoj domeni, ne moramo puno promisliti da bi zaključili da ćemo naići na problem. Naime ako se trenutno nalazimo u trenutku t, a slijedeći trenutak je $t + dt$, gdje je dt konačan broj (a ne beskonačno malen), dobivamo prekid u našoj vremenskoj krivulji i gubimo informacije unutar tog intervala. Početno rješavanje problema bi možda bilo da definiramo dt tako malen da zahtjeva sve naše potrebe, no to nije toliko robusno i optimalno, te ne može pokriti sve slučajeve. Može nam se desit na primjer da želimo definirati simulaciju sudara objekta koji putuje velikom brzinom i statičnog tijela, na primjer zida. Pošto tijelo putuje velikom brzinom, toliko velikom da u trenutno t bude ispred zida, a u trenutku $t + dt$ bude iza zida, te tako prođe kroz njega, a da se niti ne zabilježi sudar. Navedeni problem se još naziva i tuneliranje. No, vratimo se na naš prvobitni problem ne-penetracije tijela. Pošto nam spomenuto rješenje nije riješilo problem preostaje nam da se suočimo s problemom definirajući dodatne algoritme. Numerička metoda nazvana bisekcijom će nam riješiti problem što nam dt nije beskonačno malen broj. Ukoliko u trenutku t ne postoji penetracija, a u trenutku $t + dt$ zabilježimo penetraciju, ne možemo kazati sa sigurnošću da je $t + dt$ trenutak sudara. Zbog toga resetiramo integrator na vrijeme t, te napravimo skok na $t + dt/2$. Ukoliko se u tom intervalu ne događa penetracija, gledamo interval od $t + dt/2$ do $t + dt$, inače gledamo interval od t do $t + dt/4$. I tako sve dok ne dobijemo vremenski interval između t, odnosno vremena u kojem nema penetracije i nekog vremena t_s koje označava vrijeme penetracije, odnosno sudara, s time da je $t_s - t < \in$, gdje je $\in$ definirana granica tolerancije greške. Ukoliko bi odmah na početku vrijednost $\in$ postavili za dt, očigledno da se ta simulacija ne bi mogla odvijati u stvarnom vremenu. Slika 1.6 i slika 1.7 grafički prikazuju spomenuti problem.



Slika 1.6 Problem određivanja vremena sudara



Slika 1.7 Određivanje vremena sudara s tolerancijom greške $\in$

1.2.2. Kontakt između tijela

Općenito imamo dvije vrste kontakta između tijela: kontakt pri sudaru i mirujući kontakt.

Kontakt pri sudaru nazivamo kontakt kada su dva tijela u kontaktu u nekoj točki i imaju vektor smjera brzine okrenut jedan prema drugom. Takav kontakt zahtjeva instantnu promjenu vektora brzine. To znači da se vektor stanja tijela koji opisuje poziciju i brzinu, mijenja diskretno što se tiče brzine. Drugim riječima ne mijenja se postepeno, već ima grub prijelaz. Pošto je to proturječno našem modelu gibanja čvrstog tijela, jer kao što je već ranije rečeno vrijedi samo za realnu domenu s postepenim prijelazima, očito da opet

imamo problem. Da bi zaobišli ovaj problem svaki put kada se zabilježi sudar u vremenu t_c na trenutak ćemo zaustavit integrator. Uzet ćemo vektor stanja u vremenu t_c , $\boldsymbol{X}(t_c)$ i izračunat ćemo vektore brzine na temelju sudara. Novi vektor stanja $\boldsymbol{X}(t_c)'$ postavit ćemo tijelu, te ponovno pokrenut integrator. Važno je napomenuti da su vektor $\boldsymbol{X}(t_c)$ i $\boldsymbol{X}(t_c)'$ jednaki po prostornim komponentama (pozicija i orijentacija), te se razlikuju samo u komponenti brzine.

Kada je jedno tijelo položeno na drugo (brzina je jednaka nuli), kažemo da su tijela u mirujućem kontaktu. U slučaju kada imamo takav kontakt, izračunavamo silu koja sprječava da tijelo ide prema dolje. U raznim literaturama tu silu možemo naći pod nazivom sile podloge, a izračunavamo ju zbog utjecaja gravitacije (i/ili neke druge sile), koja tjera tijelo prema dolje, kako bi se poništile. Očigledno da je razrješavanje mirujućeg kontakta jednostavnije od kontakta pri sudara, te ne trebamo zaustaviti integrator, već je samo potrebno izračunati sile.

2. Simulacija fizike čvrstih tijela u stvarnom vremenu

Prvo poglavlje ovog rada govori o matematičkom modelu koji opisuje gibanje čvrstog tijela. Model je dosta opsežan, te iziskuje dosta računanja kojeg si ne možemo priuštiti u simulaciji sa stvarnom vremenu bez dodatnih optimizacija. Stoga će ovo poglavlje biti posvećeno gotovim sustavima za simulaciju fizike koji omogućuju simulaciju fizike čvrstih tijela u stvarnom vremenu. U [9, 10, 11, 12] se mogu naći slični pristupiti problemu.

2.1. SPE sustav za simulaciju fizike

SPE (*eng. Simple Physics Engine*) je djelo Phylar Laba. Prva službena prezentacijska aplikacija izdana je 2006. godine, a potom i prvi službeni SDK (verzija 1.0) 2007. godine. Od tada do danas je izdano mnoštvo verzija SDK-a. Poboljšanja su vidljiva u optimizaciji algoritama, te i u dodavanju novih funkcija. Trenutno je aktivna verzija 3.0, koju ćemo ovdje i detaljnije opisati. Stoga krenimo sa najvažnijim funkcijama koje jezgra pruža.

2.1.1. Mogućnosti jezgre

Jezgra pruža detekciju sudara objekata. To možda ne zvuči ništa posebno, pošto je to osnovna funkcionalnost za takvu jezgru. Ono što zvuči posebno je da se jezgra brine u potpunosti za detekciju sudara objekata, u smislu da joj se preda mreža poligona objekta i to je sve što trebamo napraviti. Ne trebamo se brinuti da li je objekt konveksan ili konkavan. Objekt može izraditi na temelju raznih datoteka u kojima su objekti zapisani, ali i direktno iz *vertex* i *index* spremnika. Objektu možemo dodati razne atribute kao što su masa objekta za realističnu simulaciju i drugo. Praktičnost je ovdje za programere posebno dobro realizirano.

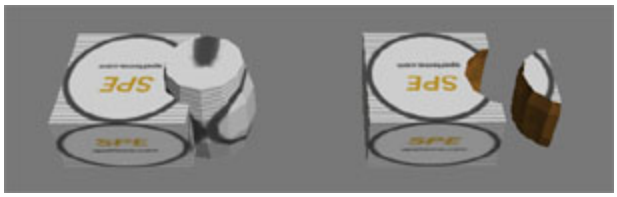
Conservative Direct Solver (CDS) je ime glavnog jezgrinog rješavača. Relativno je visoke preciznosti, te može razrješavati velike količine kontakata između objekata. Moguće je podesiti preciznost i brzinu.

Podržanost zglobova (eng. *joints*). O zglobovima ćemo više pričati u poglavlju (2.2) o ODE sustavu za simulaciju fizike, no zasad možemo reći da su to spojnice između dva

objekta. SPE podržava zglobove kojima možemo definirati maksimalnu udaljenost objekta i maksimalnu silu koji zglob podnosi prije puknuća. Phylar Lab tvrdi da postoje sustavi za simulaciju fizike (vjerojatno ODE sustav za simulaciju fizike) koji imaju bolju podržanost zglobova, ali pošto je SPE fleksibilna platforma može se lako putem matrica unositi postavke zglobova iz drugih jezgara.

Lomljenje čvrstih tijela. Ova funkcija je prva što upada u oko ukoliko radite simulaciju lomljenja čvrstih objekata. Iako još u beta fazi, pruža dosta dobre rezultate. Radi na načelu da SPE pruža operacije nad objektom tako da ga reže na temelju ravnine, kolekcije ravnina ili na temelju drugog objekta. Slika 2.1 i slika 2.2 prikazuju spomenuto.

Slika 2.1 Rezanje objekta pomoću dvije ravnine



Slika 2.2 Rezanje objekta na temelju drugog objekta

Optimirano paralelno računanje. Kako bi se iskoristili više-jezgreni procesori, SPE raspoređuje računanje po jezgrama. Phylar Lab tvrdi da preko 90 % posla se može rasporediti na različite dretve, te da uspoređujući sa jednom dretvom, dvije pružaju barem 80 % bolje performanse za integrator i detekciju sudara, te 30 % bolje performanse za rješavač.

Simulacija čestičnih sustava. SPE podržava simulaciju vode i dima, međudjelovanje s čvrstim tijelima, dinamično grupiranje, brza diskretna generacija površine i drugo.

Zadnja, no ne i najmanje važna odlika SPE-a je što ima vrlo intuitivno i lagano za koristiti sučelje.

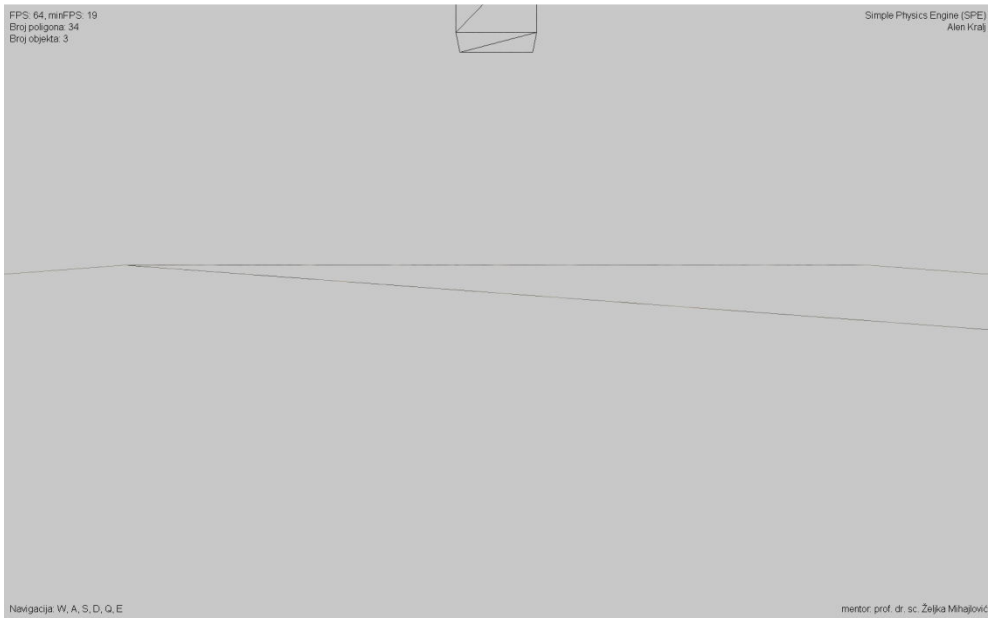
2.1.2. Aplikacija simulacije lomljenja čvrstih tijela

U ovom poglavlju prikazat ćemo rezultate korištenja SPE sustava za simulaciju fizike. Dodatak A: Dokumentacija aplikacije simulacije lomljenja čvrstih tijela govori više o načinu implementacije aplikacije, sceni i objektima, te o načinu postavljanja aplikacije.

Izvršavanje aplikacije ćemo provodit na računalu sa Athlon 64 3000+ procesorom, 1 GB RAM memorije, te sa ATI 1900GT grafičkim procesorom.

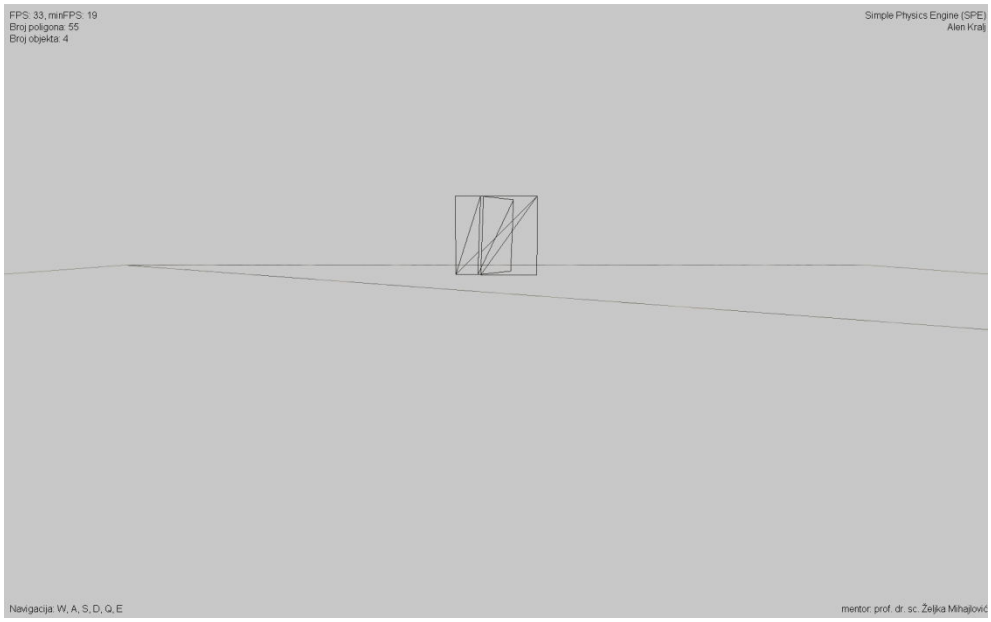
Primjer 1.

Cilj ovog primjera je pokazati kako aplikacija lomi objekte. Kako bi što zornije prikazali što se događa sa objektima postavit ćemo žičano iscrtavanje. U scenu postavljamo jednu poveću statičnu ploču na koju će padati dva dinamička objekta koja su zapravo kocke. Prvu kocku postavljamo na određenu visinu u odnosu na statičnu ploču tako da ju gravitaciju privuče i izazove sudar između statične ploče i prve kocke. Drugu kocku postavljamo na malo veću visinu od prve kocke kako bi se netom nakon sudara statične ploče i prve kocke, i druga kocka pridružila sudaru. Slika 2.3 prikazuje scenu prije sudara. U sceni se trenutno nalaze 3 objekta, od kojih na ekranu vidimo samo 2 (prvu kocku i statičnu ploču). Možemo primijetiti kako se kocka sastoji od 12 trokuta.



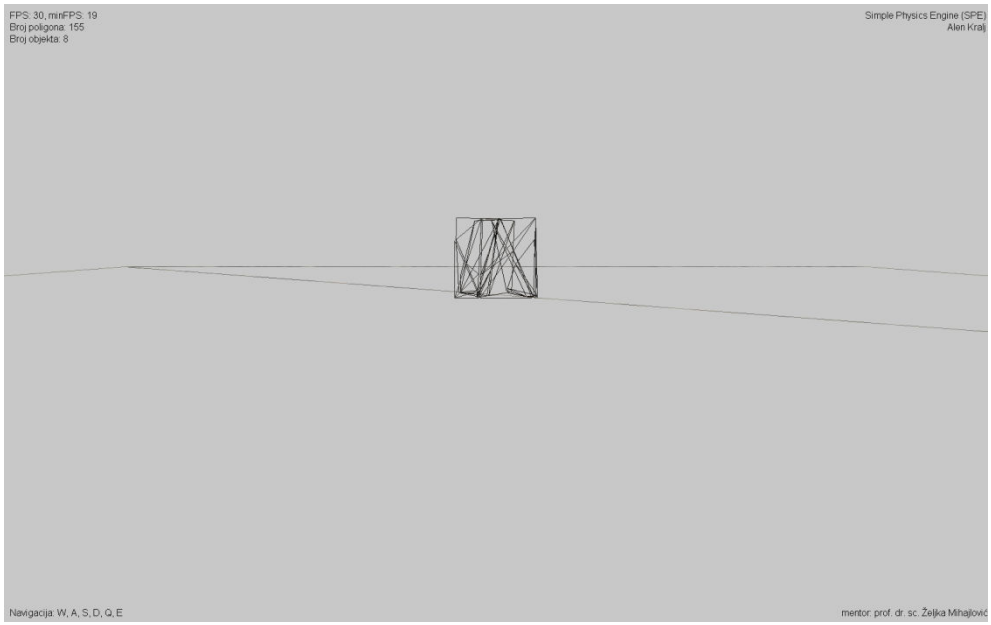
Slika 2.3 Prikaz scene prije sudara

Nakon inicijalnog sudara prve kocke i statične ploče, prva kocka se zbog siline udara malo udaljila od ploče. Sada se broj objekata u sceni povećao na 4. To je zato što se inicijalna prva kocka razlomila. Slika 2.4 prikazuje inicijalni sudar.



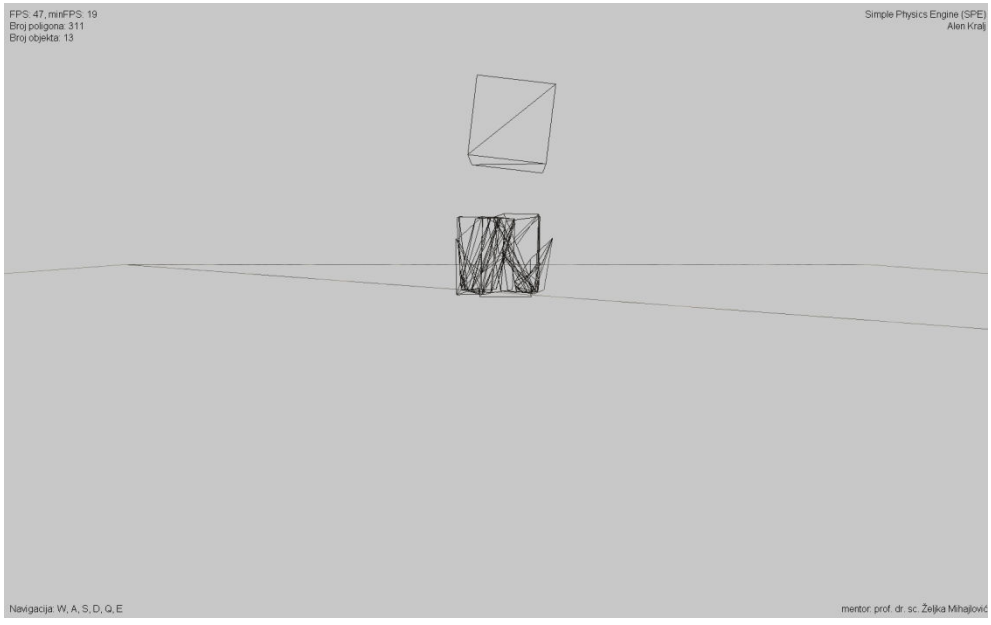
Slika 2.4 Inicijalni sudar prve kocke sa statičnom pločom

Slijedeći sudar koji će se desit je kada prva kocka ponovno padne na ploču. Slika 2.5 prikazuje ponovni sudar prve kocke sa pločom.



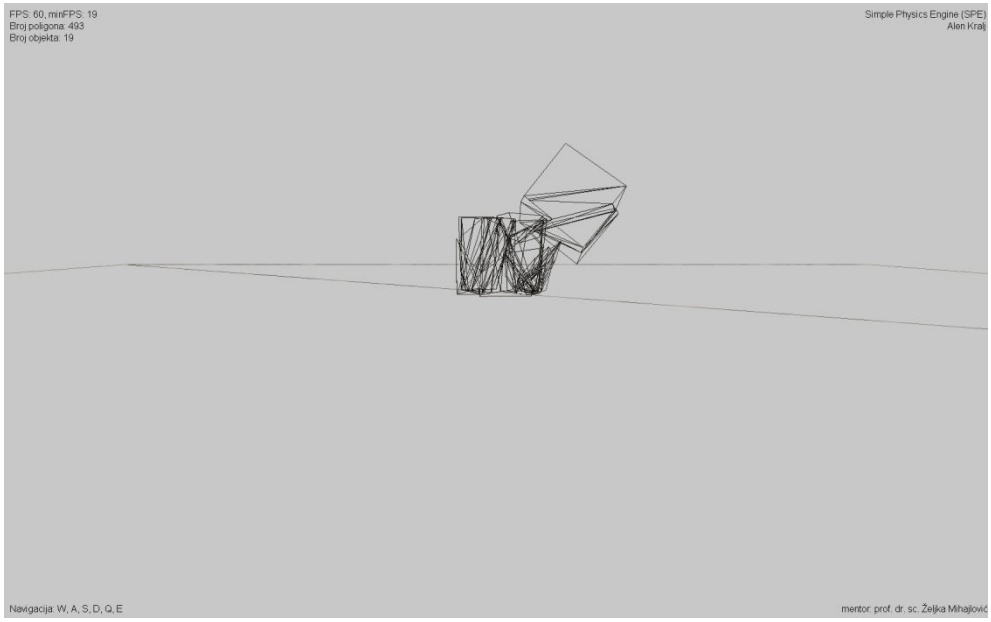
Slika 2.5 Ponovni sudar prve kocke sa pločom

Primijetimo kako se broj objekta u sceni povećao na 8. Slijedeći sudar se događa kada dolazi druga kocka, te se sudara s prvom kockom. Nakon sudara druga kocka se zbog siline udara udaljila od prve kocke. Slika 2.6 prikazuje spomenuti sudar.



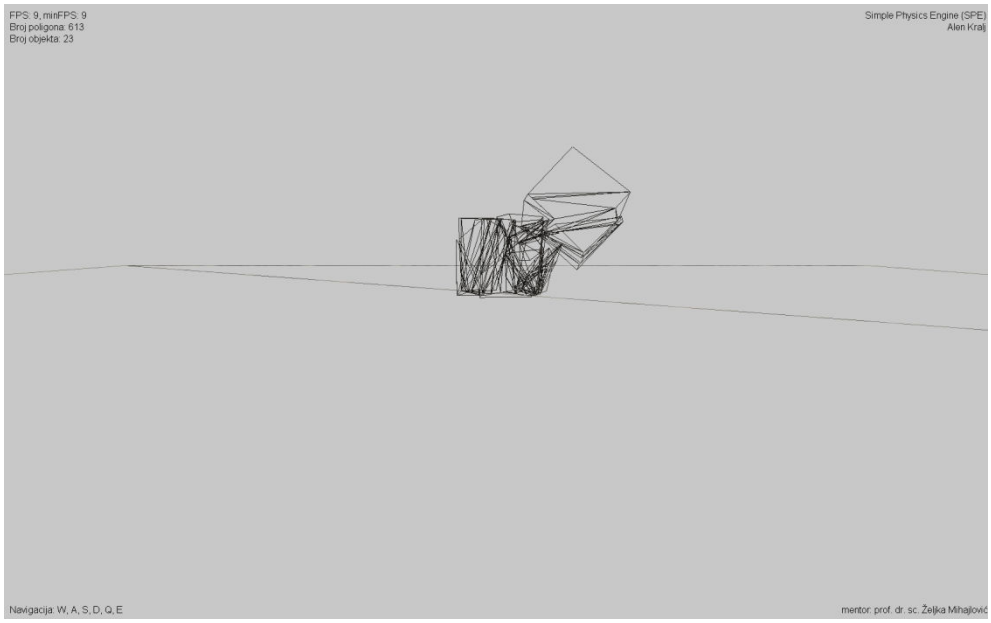
Slika 2.6 Sudar prve i druge kocke

Broj objekata u sceni je ostao isti, ali to ne mora uvijek biti tako. Slijedeći sudar je kada ponovno druga kocka padne na prvu kocku. Na slici 2.7 možemo vidjeti spomenuti sudar, te primijetimo kako se broj objekta u sceni povećao na 19, odnosno nije ostao isti kao ranije.

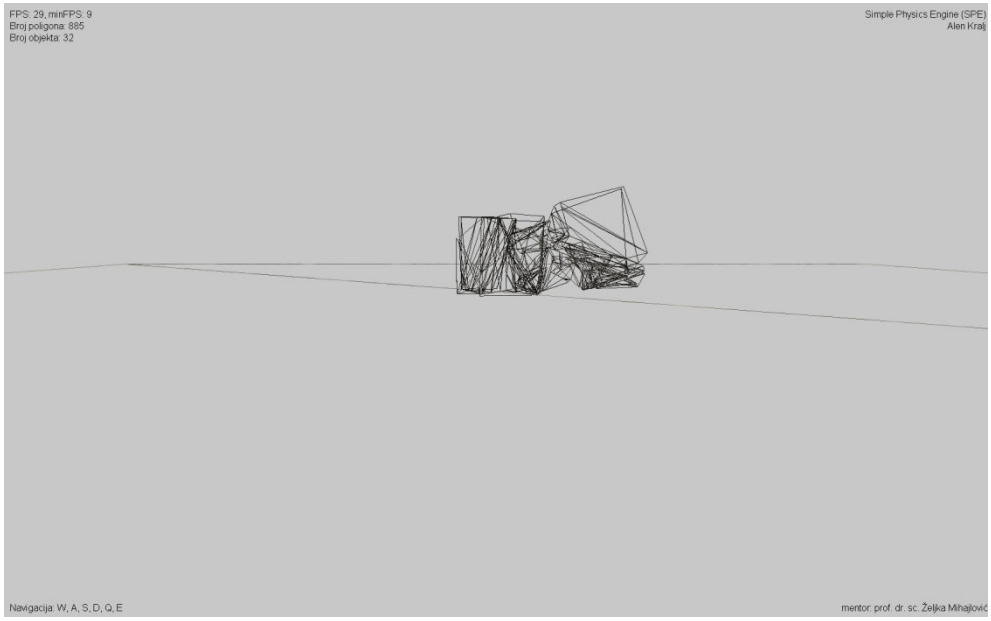


Slika 2.7 Ponovni sudar prve i druge kocke

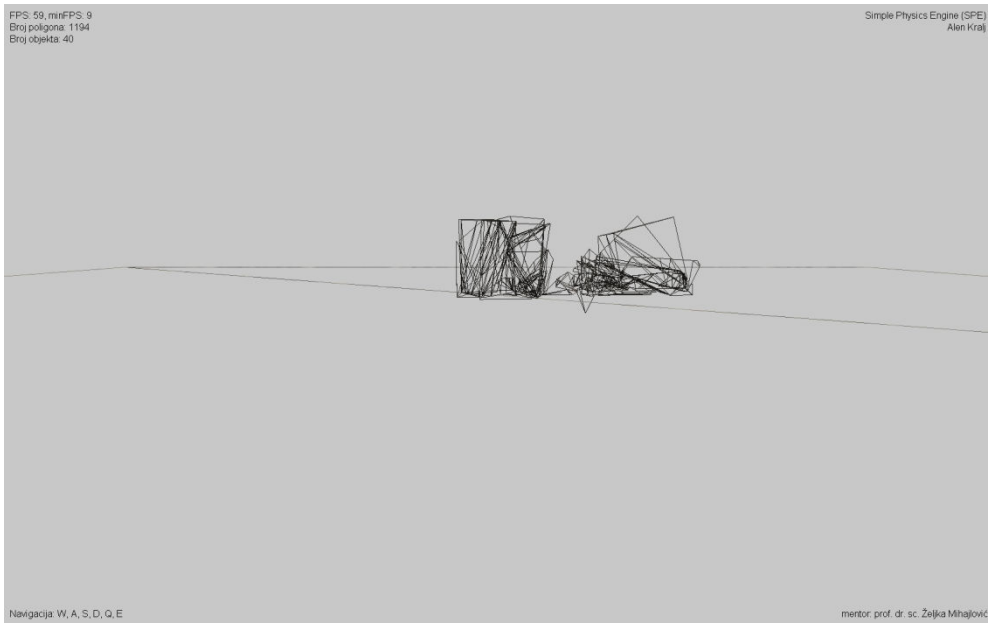
Slike od 2.8 do 2.10 prikazuju posljednje sudare u sceni.



Slika 2.8 Ponovni sudar druge kocke sa prvom (23 objekta u sceni)

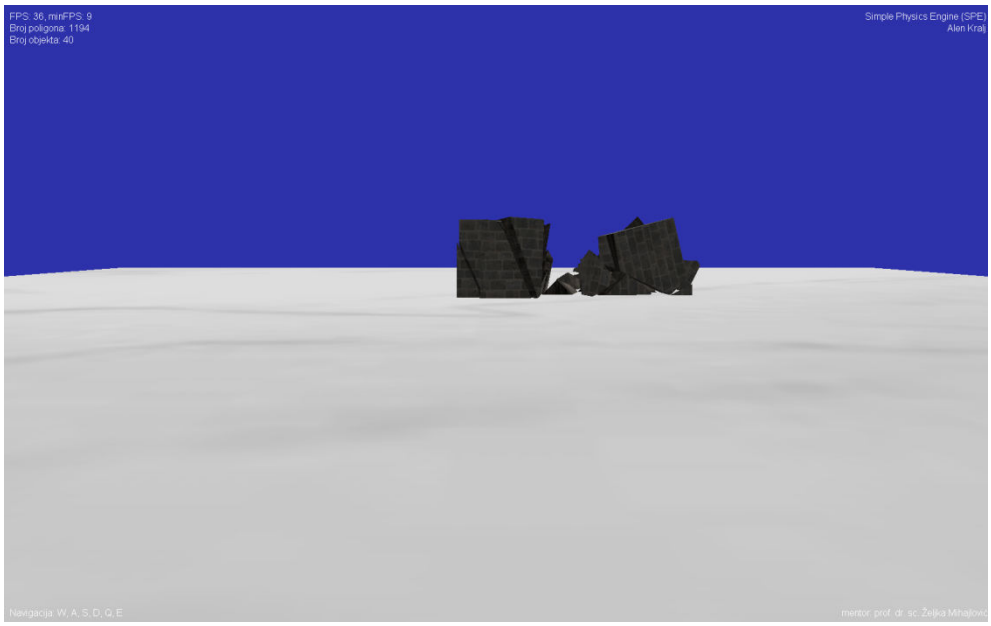


Slika 2.9 Sudar druge kocka sa pločom (32 objekta u sceni)



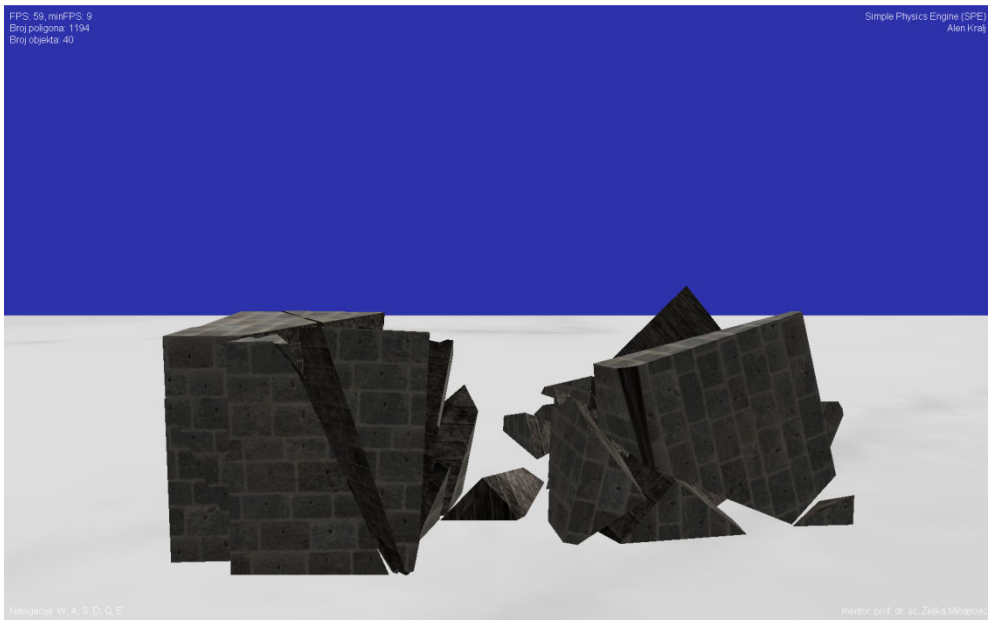
Slika 2.10 Sudar druge kocke sa pločom (40 objekta u sceni)

Ukoliko isključimo žičano iscrtavanje dobivamo sliku 2.11.



Slika 2.11 Normalno iscrtavanje (40 objekta u sceni)

Ukoliko se približimo objektima možemo vidjeti razliku između vanjske i unutarnje teksture objekta.

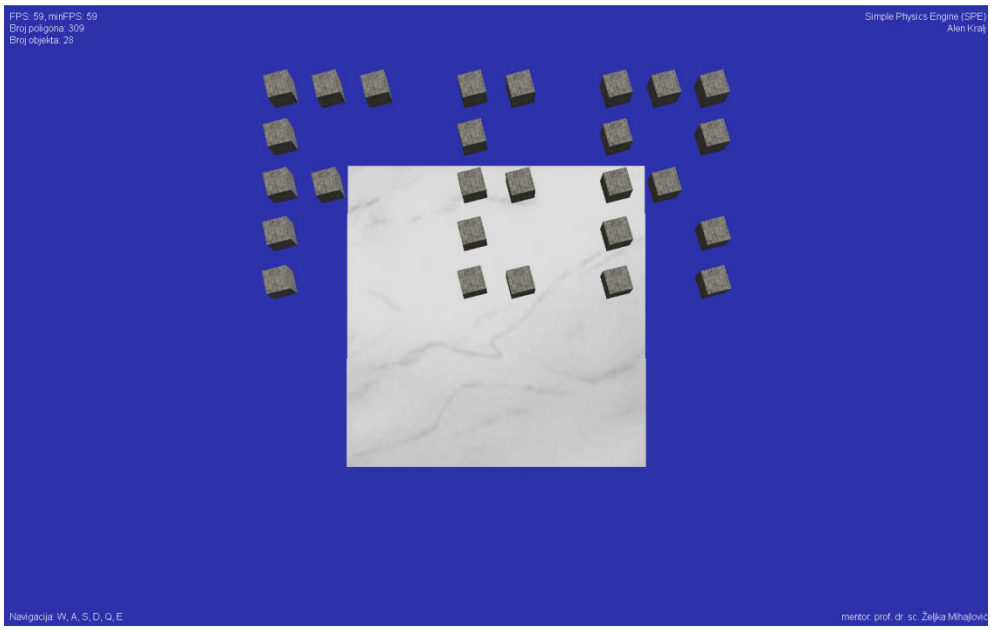


Slika 2.12 Vidljiva razlika vanjske i unutarnje teksture

Prosječna brzina iscrtavanja je 60 slika u sekundi. Pri sudarima taj broj padne na oko 30 slika u sekundi za ovako jednostavnu scenu. Ukoliko imate dobar vid možete primijetiti da zapravo na prethodnim slikama piše da je minimalni broj slika u sekundi 9. To je naravno posljedica snimanja ekrana koja dodatno opterećuje računalo.

Primjer 2.

U ovom primjeru cilj nam je prikazati kako aplikacija radi sa povećim brojem objekata. U scenu ćemo ponovno staviti poveću statičnu ploču i 27 dinamičkih objekata (kocka) koje tvore natpis poznatog fakulteta elektrotehnike i računarstva. Naknadno ćemo mijenjati postavke dinamičkih objekata, točnije rečeno mijenjat ćemo najmanji mogući volumen lomljenog tijela, te ćemo tako postići drastične razlike u broju novonastalih objekata. Slika 2.13 prikazuje scenu prije sudara.



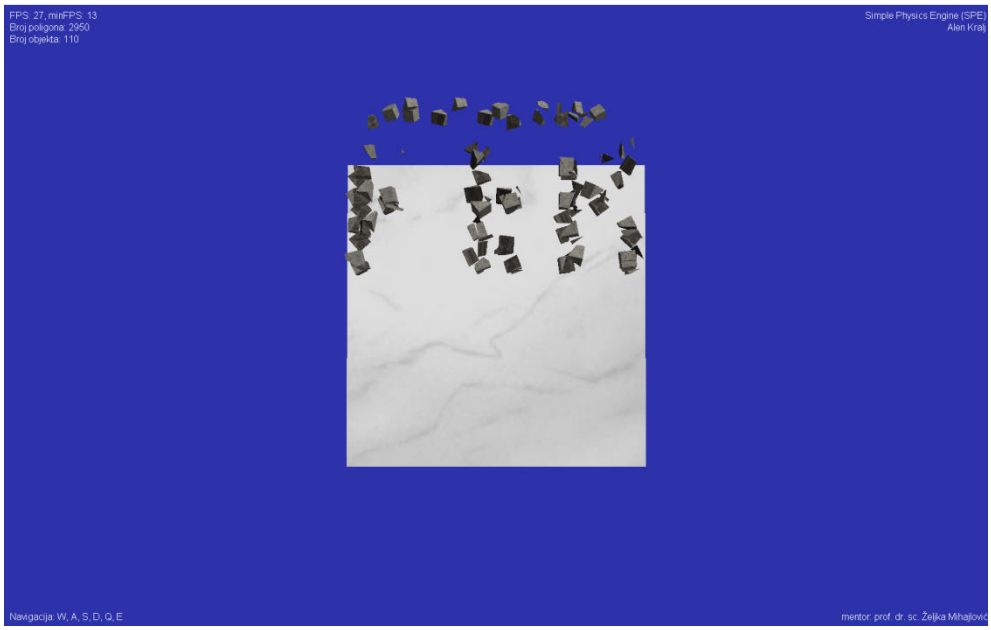
Slika 2.13 Inicijalna scena sa 28 objekta

Ukoliko postavimo dinamičkim objektima minimalni volumen lomljenih tijela na 0.05, nakon svih sudara imat ćemo u sceni oko 200 objekata, te oko 6 do 7 tisuća poligona. Slika 2.14 prikazuje spomenuto.

Slika 2.14 Scena sa objektima minimalnog volumena lomljena tijela 0.05

Ukoliko objektima postavimo minimalni volumen lomljenja tijela na 0.9, možemo vidjeti na slici 2.15 kako će se lomljenje reducirati, te ćemo imati znatno manje novonastalih

objekata. Ostale parametre objekata naravno ne mijenjamo. U konačnici će biti oko 100 novonastalih objekata, te oko 3 tisuće poligona.



Slika 2.15 Scena sa objektima minimalnog volumena lomljena tijela 0.9

Što manji minimalni volumen lomljenja tijela postavimo više ćemo novonastalih objekata dobiti. To naravno povlači i da će padati broj slika u sekundi. Pa tako dok smo postavili minimalni volumen lomljenja tijela na 0.05 prosječan broj slika u sekundi pri sudarima bio je 5, a dok smo postavili minimalni volumen lomljenja tijela na 0.9 prosječan broj slika u sekundi pri sudarima bio je 15.

2.2. ODE sustav za simulaciju fizike

ODE (*eng. Open Dynamics Engine*) je djelo Russell Smitha [14]. Prva verzija ODE SDK-a je izašla početkom 2001. godine i od onda redovito izlaze nove verzije. Ova jezgra za simulaciju fizike nam je posebno zanimljiva pošto ima mogućnost definiranja različitih zglobova. Zglob je ništa drugo nego određena povezanost objekata.

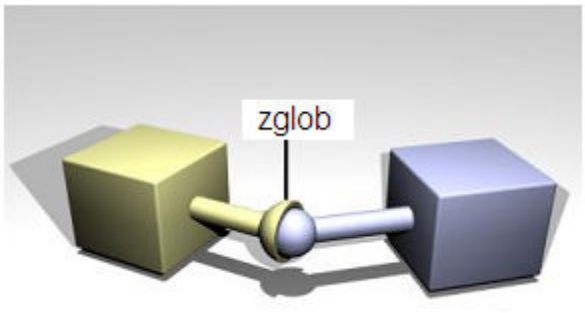
2.2.1. Zglobovi

Trenutno u ODE svijetu je moguće definirati 5 zglobova:

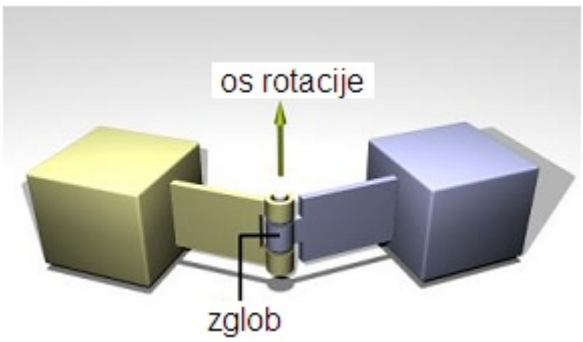
- kuglasti

- okomiti
- klizački
- univerzalni
- motorni

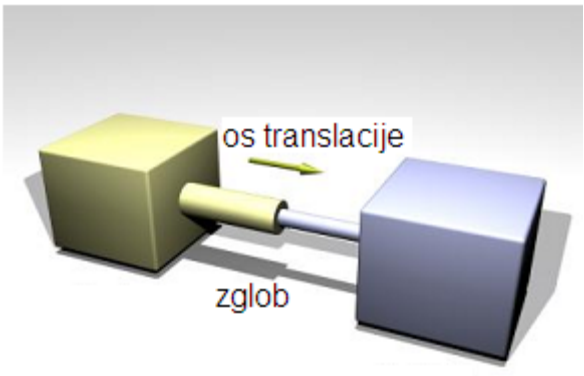
Slike od 2.16 do 2.20 prikazuju spomenute zglobove.



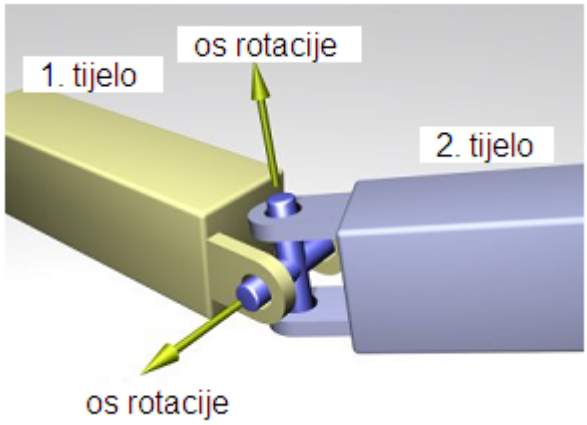
Slika 2.16 Kuglasti zglob



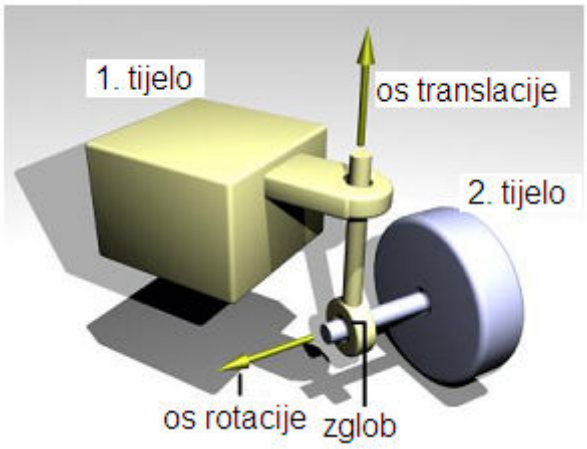
Slika 2.17 Okomiti zglob



Slika 2.18 Klizački zglob



Slika 2.19 Univerzalni zglob

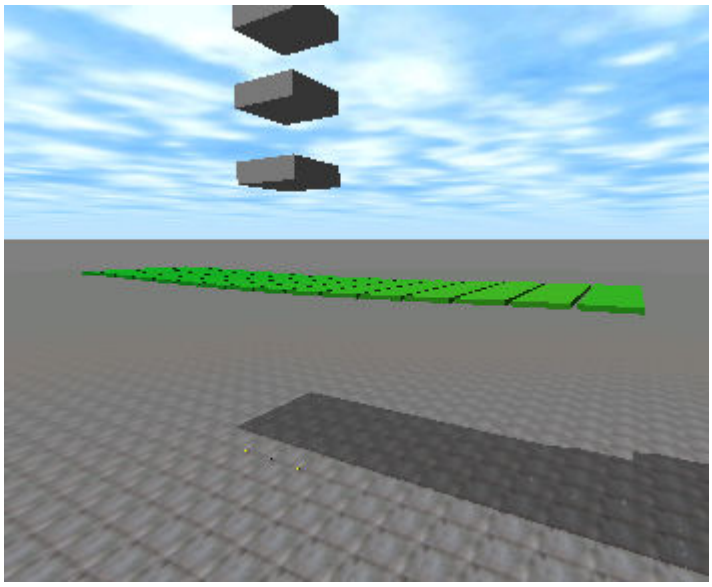


Slika 2.20 Motorni zglob

Ovisno o vrsti zgloba možemo kontrolirati ograničenja, odnosno slobodu između objekta. Osim spomenutog, zglobovima možemo podesiti silu podnošljivosti, odnosno silu pri kojom puca zglob. Ovo nam je jako zanimljivo, jer tako možemo simulirati lomljenje tijela. U poglavlju 2.1 opisali smo simulaciju lomljenja objekta, tako da ga režemo. Ovdje ćemo simulaciju lomljenja tijela prikazati tako da jedno tijela sastavimo od većeg broja manjih objekata povezanih zglobovima. Općenito takav model simulacije nazivamo metoda simulacije sa konačnim brojem elemenata.

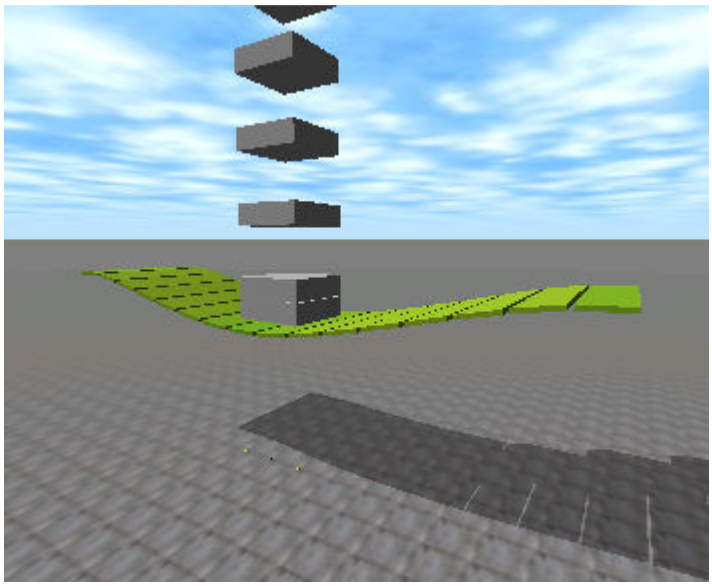
2.2.2. Aplikacija simulacije lomljenja tijela sa konačnim brojem objekata

Sa ODE SDK-om dobivamo demonstracijsku aplikaciju `demo_feedback`. Aplikacija prikazuje most sastavljen od 16 objekata (daščica) međusobno povezanih klizačkim zglobom. Iznad mosta se nalaze 10 objekata koji će pasti na most. Cilj je simulirati kako će most puknuti pri opterećenju. Potrebno je podesiti masu objekata, silu puknuća na klizačkim zglobovima i gravitaciju. U aplikaciji je to sve već podešeno, te možemo odmah prikazat rezultate. Slika 2.21 prikazuje scenu prije početka sudara.

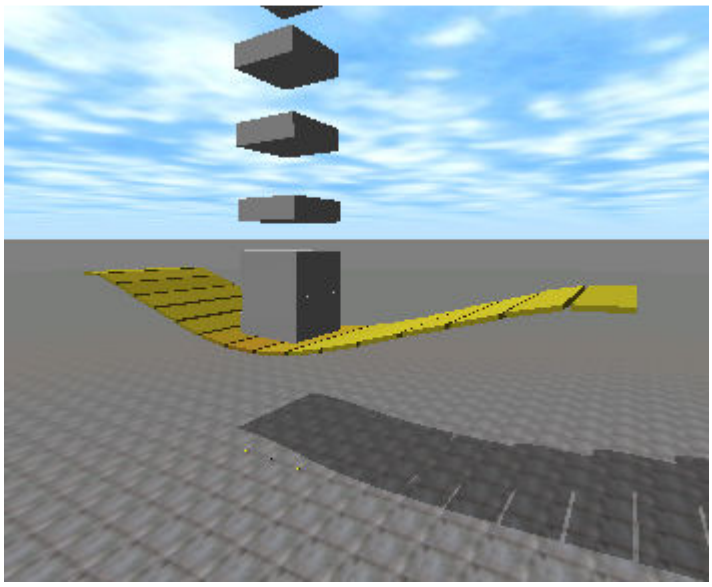


Slika 2.21 Scena prije početka sudara

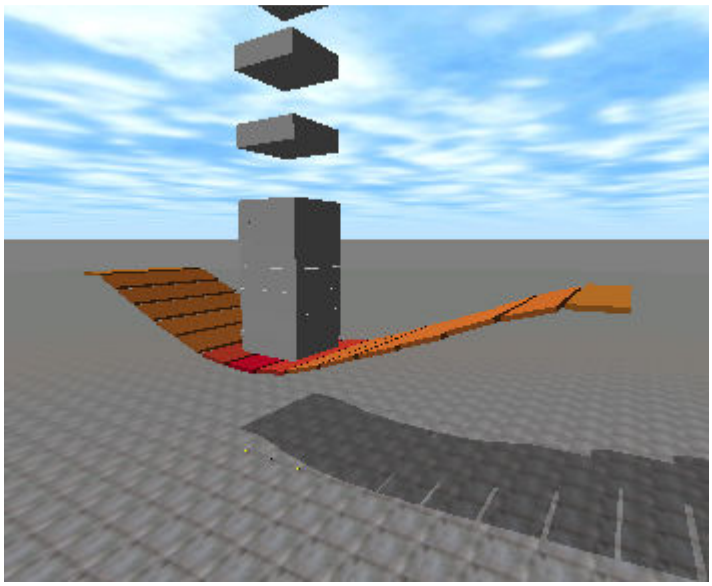
Objekti od kojih je sastavljen most mijenjaju boju od zelene do crvene u ovisnosti pod kojim su opterećenjem njihovi zglobovi. Slike 2.22, 2.23, 2.24 i 2.25 prikazuju most pod različitim opterećenjem.



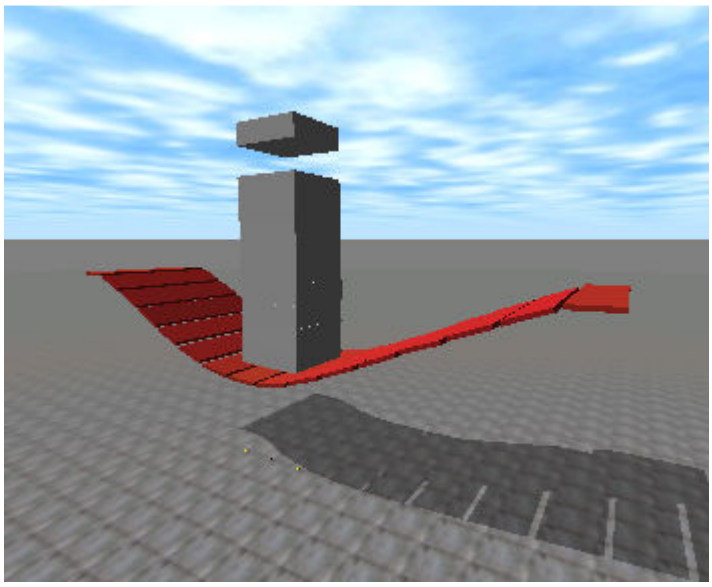
Slika 2.22 Opterećenje mosta sa 2 objekta



Slika 2.23 Opterećenje mosta sa 5 objekta

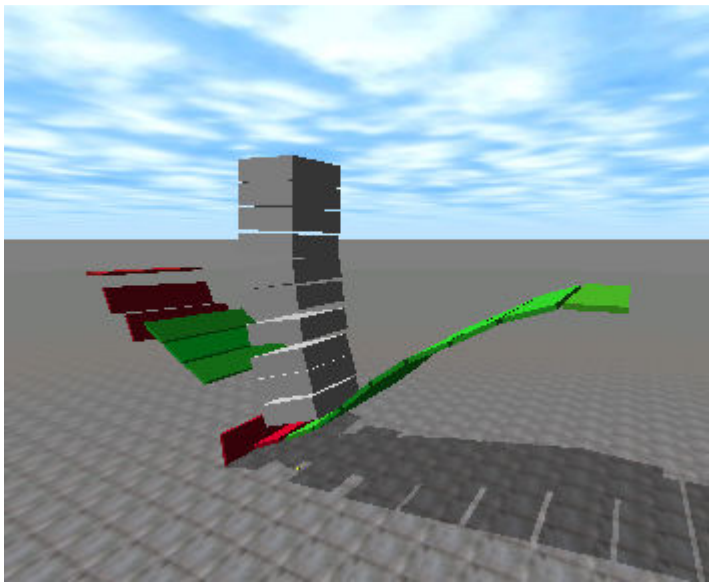


Slika 2.24 Opterećenje mosta sa 7 objekta



Slika 2.25 Opterećenje mosta sa 9 objekta

Te na kraju kada zglobovi više ne mogu podnijeti silu objekata dolazi do puknuća. Slika 2.26 prikazuje puknuće mosta.



Slika 2.26 Puknuće mosta

Za razliku od metode simulacije lomljenja tijela u poglavlju 2.1, ova metoda se može pohvaliti značajno manjim zahtjevima za resurse. Ali s druge strane i značajnim ograničenjima. Broj slika u sekundi je konstantan, odnosno kada ima sudara i kada nema sudara objekata ne dolazi do znatne promjene u broju slika u sekundi.

2.3. BPL sustav za simulaciju fizike

BPL (*eng. Bullet Physics Library*) je još jedan profesionalni sustav za simulaciju fizike [15]. Autor je Erwin Coumans, bivši zaposlenik najpoznatijeg komercijalnog sustava za simulaciju fizike *Havok*. Prva verzija sustava izašla je 2003. godine, a od 2005. sustav je otvorenog kôda. Osim PC platforme može se koristiti i na slijedećim konzolama: *Play Station 2, Play Station 3, X Box 360 i Nintendo Wii*. Pošto je otvorenog kôda Sony je dodao optimizaciju za Play Station 3, odnosno njegov *Cell SPU* (*eng. Synergic Processing Unit*) procesor, te tako postiže dobre rezultate. Korišten je mnogim igrama.

Glavne mogućnosti jezgre su:

- detekcija kolizije sa konkavnim i konveksnim tijelima
- dinamika čvrstih tijela
- zglobovi
- simulacija vozila
- prevođenja kôda za sve platforme

- više-jezgrena optimizacija

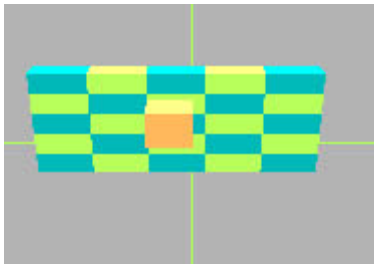
Iako sustav podržava znatne funkcionalnosti, ovdje ih nećemo prikazivati, već ćemo objasniti još jedan (jednostavan) način lomljenja tijela.

2.3.1. Aplikacija simulacije jednostavnog lomljenja tijela

U poglavlju 2.2 vidjeli smo zanimljiv način simuliranja lomljenja tijela. Ovdje ćemo tu ideju pojednostaviti tako da ćemo izbaciti zglobove. Naravno onda ne možemo simulirati lomljenje mosta, ali recimo rušenje zida možemo. Ideja je da zid sastavimo od nekoliko objekata. Konkretno, ovdje ćemo to napraviti tako da je jedan objekt jedna ciglica. Prilikom gađanja zida drugim objektom želimo da se zid što stvarnije sruši. Koristit ćemo BPL-ovu demo aplikaciju `basicdemo` koju ćemo prepravit po potrebi.

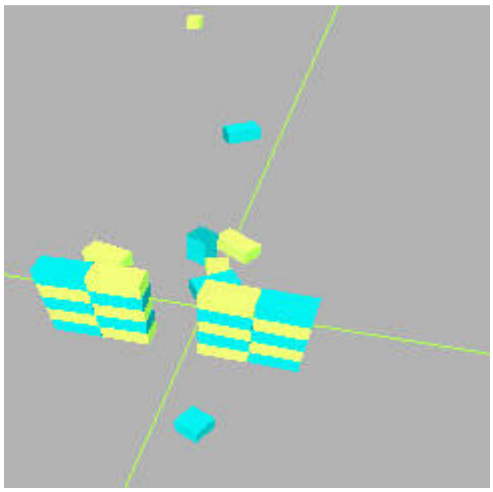
Primjer 1.

Složimo zid od nekoliko manjih objekata, ciglica kao što je na slici 2.27.



Slika 2.27 Jednostavan zid

Zelenim i plavim bojama su označene ciglice zida, dok je narančasto obojan objekt kojim ćemo pokušat srušit zid. Slika 2.28 prikazuje posljedicu sudara objekta sa zidom.

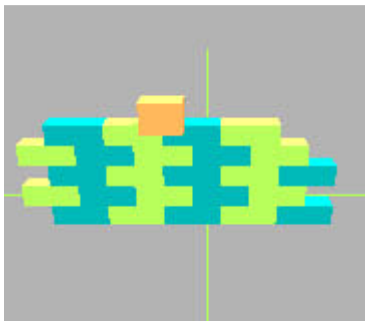


Slika 2.28 Rušenje jednostavnog zida

Vidimo da rezultat nije baš kako smo se nadali. Pošto ciglice nisu međusobno povezane zglobovima, ukoliko napadnemo jedan stupac ciglica, susjedni stupci se ne ruše što nije ispravno. Naime, možemo primijetiti jednu zanimljivu pojavu. Ako napadnemo jednu ciglicu unutar stupca (čak i onu najvišu) da će to imati posljedicu i na ostale ciglice u stupcu. Razlog tome je što su ciglice ipak povezane na neki način, jer sustav ima podršku za simulaciju trenja. Ukoliko želimo u potpunosti iskoristiti tu ideju moramo drugačije sagraditi zid.

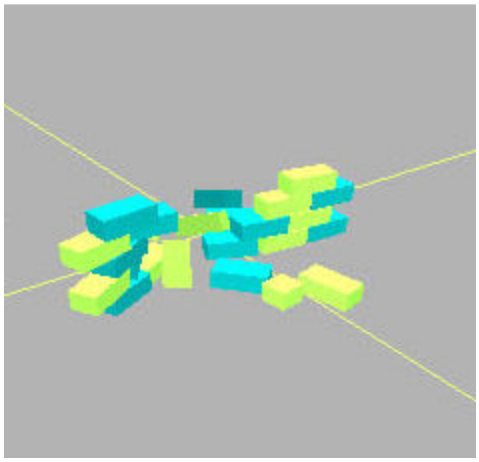
Primjer 2.

Pokušajmo ponovno složiti zid od ciglica, baš kako zidari slažu zid od pravih cigala. Slika 2.29 prikazuje takav zid.



Slika 2.29 Stvarni zid

Sada kada napadnemo takav zid sa objektom dobit ćemo puno bolje rezultate. Slika 2.30 prikazuje rezultat rušenja stvarnog zida.



Slika 2.30 Rušenje stvarnog zida

Zaključak

Osnovno što je potrebno da bi se ostvarila simulacija lomljenja čvrstih tijela je sustav za simulaciju fizike. Prvo poglavlje ovog rada daje uvid u jednostavan matematički model mehanike. Iako jednostavan, pruža mnoge mogućnosti, ali i stvara mnoge probleme pri implementaciji na računalu. Kada se prebrodi sustav za simulaciju fizike i svi njegovi implementacijski problemi, slijedi drugo poglavlje ovog rada koje pruža uvid u tri načina simuliranja lomljenja tijela.

Prvi, ujedno i najkompliciraniji način lomljenja je ostvaren tako da se režu objekti pomoću ravnine. Rezultati su pokazali da metoda znatno iziskuje performanse računala, ali ujedno daje i dobre rezultate.

Drugi način bio je utemeljen na metodi simulacije sa konačnim brojem objekata koji su međusobno povezani zglobovima. Ovdje su rezultati pokazali kako nije potrebno toliko računanja da bi postigli takvu simulaciju, te tako jako malo opterećujemo računalo.

Treći, ujedno i najjednostavniji način, bio je nastavak drugog načina tako da se izbacuju zglobovi. Pošto sustav za simulaciju fizike i bez zglobova može dati prilično dobre rezultate za određene simulacije lomljenja čvrstih tijela iskorištavamo tu ideju i pojednostavljujemo model.

U pravilu sva tri načinu su dobra. Ključ uspjeha je korištenje određenog načina za njemu prikladan scenarij. Pošto treći način najmanje opterećuje računalo, njega ćemo radije odabrati umjesto prvog ili drugog, ako za isti scenarij možemo birati između sva tri. Isto tako radije odabiremo drugi način ako biramo između prvog ili drugog. Neke stvari s druge strane ne možemo ili ne želimo simulirati sa drugim ili trećim načinom, pa biramo prvi način pošto on daje najbolje rezultate, ali nažalost na cijenu resursa. S takvom organizacijom možemo imati puno lomljenja čvrstih tijela u sceni, a da se pritom sve izvršava u stvarnom vremenu i naravno ono što je najvažnije da je zadovoljavajućeg izgleda.

Literatura

[1] BARAFF, D: “An Introduction to Physically Based Modeling: Rigid Body Simulation I: Unconstrained Rigid Body Dynamics”, *SIGGRAPH*, 1997.

[2] BARAFF, D: “An Introduction to Physically Based Modeling: Rigid Body Simulation II: Nonpenetration Constraints”, *SIGGRAPH*, 1997.

[3] HORVAT, D: “Fizika I: mehanika i toplina”, *HINUS*, Zagreb, 2005.

[4] WITKIN, A: “An Introduction to Physically Based Modeling: Particle System Dynamics”, *SIGGRAPH*, 1997.

[5] EGAN, K: “Techniques for Real-Time Rigid Body Simulation”, Providence, Rhode Island, 2003.

[6] KASS, M: “An Introduction to Physically Based Modeling: An Introduction to Continuum Dynamics for Computer Graphics”, *SIGGRAPH*, 1997.

[7] WITKIN, A; BARAFF, D: “An Introduction to Physically Based Modeling: Differential Equation Basics”, *SIGGRAPH*, 1997.

[8] WITKIN, A: “An Introduction to Physically Based Modeling: Constrained Dynamics”, *SIGGRAPH*, 1997.

[9] MÜLLER, M; MCMILLAN, L; DORSEY, J; JAGNOW, R: “Real-Time Simulation of Deformation and Fracture of Stiff Materials”, *Laboratory for Computer Science, Massachusetts Institute of Technology*, 2001.

[10] MARTINET, A; GALIN, E; DESBENOIT, B; AKKOUCHE S: “Procedural Modeling of Cracks and Fractures”, *Proceedings of the Shape Modeling International, IEEE Computer Society*, 2004

[11] MOR, A. B: ”Progressive Cutting with Minimal New Element Creation of Soft Tissue Models for Interactive Surgical Simulation”, doktorski rad, Robotics Institue, Carnegie Mellon University, 2001.

[12] O’BRIEN, J. F; HODGINS, J. K: ”Graphical Modeling and Animation of Brittle Fracture”, *SIGGRAPH*, 1999.

[13] BAO, Z; HONG, J; TERAN, J; FEDKIW, R: ”Fracturing Rigid Materials”, *IEEE Transactions on Visualization and Coputer Graphics*, 2007.

[14] SMITH, R: ”Open Dynamics Engine User Guide”, verzija 0.5, veljača 2006.

[15] COUMANS, E: ”Bullet Physics Library User Manual”, veljača 2008.

Dodatak A: Dokumentacija aplikacije simulacije lomljenja čvrstih tijela

Ovo poglavlje detaljno opisuje aplikaciju priloženu uz ovaj rad. Za postupke iscrtavanja koristit ćemo Direct3D, točnije DXUT (DirectX Utility Toolkit). DXUT nam omogućuje jednostavno pisanje aplikacije, te ga ovdje koristimo kako bi minimizirali kôd koji se brine za iscrtavanje scene. DXUT kod DirectX platforme je analogno što i GLUT kod OpenGL platforme. Stoga imamo slijedeće funkcije povratnog poziva:

```
bool CALLBACK IsDeviceAcceptable( D3DCAPS9* pCaps, D3DFORMAT
AdapterFormat, D3DFORMAT BackBufferFormat, bool bWindowed, void*
pUserContext );


HRESULT CALLBACK OnResetDevice( IDirect3DDevice9* pd3dDevice, const
D3DSURFACE_DESC* pBackBufferSurfaceDesc, void* pUserContext );


bool CALLBACK ModifyDeviceSettings( DXUTDeviceSettings*
pDeviceSettings, const D3DCAPS9* pCaps, void* pUserContext );


void CALLBACK OnFrameMove( IDirect3DDevice9* pd3dDevice, double fTime,
float fElapsedTime, void* pUserContext );


void CALLBACK OnFrameRender( IDirect3DDevice9* pd3dDevice, double
fTime, float fElapsedTime, void* pUserContext );


void CALLBACK OnLostDevice( void* pUserContext );


void CALLBACK OnDestroyDevice( void* pUserContext );


void CALLBACK OnKeyboard( UINT nChar, bool bKeyDown, bool bAltDown,
void* pUserContext );
```

Glavne metode aplikacije

Funkcija `IsDeviceAcceptable` provjerava da li je adapter (grafička kartica) uspješno dohvaćena, te da li pruža mogućnosti kao što je prozirnost, provjerava veličinu spremnika i drugo. Rezultat funkcije je `true` ukoliko adapter zadovoljava specifikacije, inače `false`.

```
bool CALLBACK IsDeviceAcceptable( D3DCAPS9* pCaps, D3DFORMAT
AdapterFormat, D3DFORMAT BackBufferFormat, bool bWindowed, void*
pUserContext )
{

    IDirect3D9* pD3D = DXUTGetD3DObject();
    if( FAILED( pD3D->CheckDeviceFormat( pCaps->AdapterOrdinal,
                  pCaps->DeviceType, AdapterFormat,
                  D3DUSAGE_QUERY_POSTPIXELSHADER_BLENDING,
                  D3DRTYPE_TEXTURE, BackBufferFormat ) ) )
        return false;

    return true;
}
```

Funkcija `OnCreateDevice` služi kao inicijalizacijska funkcija, te postavlja kameru (očiste i gledište), inicijalizira SPE svijet, postavlja gravitaciju, postavlja objekte u sceni, te postavlja način iscrtavanja (normalno ili žičano).

```
HRESULT CALLBACK OnCreateDevice( IDirect3DDevice9* pd3dDevice, const
D3DSURFACE_DESC* pBackBufferSurfaceDesc, void* pUserContext )
{
      eye = new D3DXVECTOR3(configParser->GetEyePositionX(),
                            configParser->GetEyePositionY(),
                            configParser->GetEyePositionZ());

      lookAt = new D3DXVECTOR3(configParser->GetLookAtPositionX(),
                               configParser->GetLookAtPositionY(),
                               configParser->GetLookAtPositionZ());

      viewUp = new D3DXVECTOR3(0.0f, 1.0f, 0.0f);

      D3DXMatrixLookAtLH( &matView, eye, lookAt, viewUp);

      pWorld=CreateSPEWorld();
      pWorld->SetGravity(SPEVector(0, -9.8f, 0));

      SetObjects(pd3dDevice);

      LoadFont(pd3dDevice);

      if (configParser->GetWireframe())
      {
            pd3dDevice->SetRenderState(D3DRS_FILLMODE,
                                             D3DFILL_WIREFRAME);
      }

    return S_OK;
}
```

Funkcija `OnResetDevice` nam služi za postavljanje kuta gledanja (*eng. field of view*).

```
HRESULT CALLBACK OnResetDevice( IDirect3DDevice9* pd3dDevice,
                                const D3DSURFACE_DESC*
pBackBufferSurfaceDesc, void* pUserContext )
{
      float fAspectRatio = pBackBufferSurfaceDesc->Width /
                           (float)pBackBufferSurfaceDesc->Height;

      D3DXMatrixPerspectiveFovLH(&matProj, D3DX_PI/4, fAspectRatio,
                                 0.1f, 1000.0f );
    return S_OK;
}
```

Funkcija `OnFrameMove` nam služi kako bi osvježili scenu. Pozivamo SPE svijet, kako bismo mu javili koliko je vremena prošlo od zadnjeg osvježavanja, te pozivamo funkciju za provjeru lomljenja tijela.

```
void CALLBACK OnFrameMove( IDirect3DDevice9* pd3dDevice, double fTime,
float fElapsedTime, void* pUserContext )
{
      pWorld->Update(fElapsedTime);
      CheckBreak();
}
```

Funkcija `OnFrameRender` služi za iscrtavanje scene i sučelja. Pošto je poveća ovdje ćemo istaknut samo najbitnije stvari, a to je kako iscrtavamo objekte. Točnije kako ljepimo teksture na objekt. Pošto objekt može imati drugačiji izgled iznutra nego izvana, potrebno je lijepiti unutarnju teksturu na površine koje su rezane (novonastale površine), a vanjsku teksturu na površine koje nisu. Statični objekti ne mogu biti lomljeni, te njih lakše iscrtavamo, odnosno samo ljepimo vanjske teksture.

```
for (int i = 0; i < Bodies.size; i++)
{
      Bodies[i]->GetTransformMesh(&matWorld);
      pd3dDevice->SetTransform(D3DTS_WORLD, &matWorld);

      if (Bodies[i]->GetBeStatic())
      {
            ObjectUserData* userData =
                        (ObjectUserData*)Bodies[i]->UserData;
            LPD3DXMESH pMesh=(LPD3DXMESH)(userData->pMesh);
            pd3dDevice->SetTexture(0,
                        textures[userData->outsideTextureId]);
            pMesh->DrawSubset(0);

            faceCount += pMesh->GetNumFaces();
      }
      else
      {
            ObjectUserData* userData =
                        (ObjectUserData*)Bodies[i]->UserData;
            SPEMesh* pMesh=Bodies[i]->GetShape()->GetMesh();

            pd3dDevice->SetTexture(0,
                        textures[userData->outsideTextureId]);
            pd3dDevice->SetTexture(1,
                        textures[userData->insideTextureId]);
            pd3dDevice->DrawPrimitiveUP(D3DPT_TRIANGLELIST,
                  pMesh->GetNum(0), pMesh->GetData(0),
                  sizeof(SPEVertex));

            pd3dDevice->SetTexture(0,
                        textures[userData->insideTextureId]);
            pd3dDevice->DrawPrimitiveUP(D3DPT_TRIANGLELIST,
                  pMesh->GetNum(1), pMesh->GetData(1),
                  sizeof(SPEVertex));

            faceCount += pMesh->GetNum(0) + pMesh->GetNum(1);
      }
}
```

Funkcija `OnKeyboard` služi kako bi omogućili navigaciju kamere kroz scenu.

```
void CALLBACK OnKeyboard( UINT nChar, bool bKeyDown, bool bAltDown,
                          void* pUserContext )
{
      float delta = 1.0f;

      if (nChar == 'Q')
      {
            *eye -= D3DXVECTOR3(0.0f, 1.0f, 0.0f) * delta;
            *lookAt -= D3DXVECTOR3(0.0f, 1.0f, 0.0f) * delta;
      }
      else if (nChar == 'E')
      {
            *eye += D3DXVECTOR3(0.0f, 1.0f, 0.0f) * delta;
            *lookAt += D3DXVECTOR3(0.0f, 1.0f, 0.0f) * delta;
      }
      else if (nChar == 'W')
      {
            *eye -= D3DXVECTOR3(1.0f, 0.0f, 0.0f) * delta;
            *lookAt -= D3DXVECTOR3(1.0f, 0.0f, 0.0f) * delta;
      }
      else if (nChar == 'S')
      {
            *eye += D3DXVECTOR3(1.0f, 0.0f, 0.0f) * delta;
            *lookAt += D3DXVECTOR3(1.0f, 0.0f, 0.0f) * delta;
      }
      else if (nChar == 'A')
      {
            *eye -= D3DXVECTOR3(0.0f, 0.0f, 1.0f) * delta;
            *lookAt -= D3DXVECTOR3(0.0f, 0.0f, 1.0f) * delta;
      }
      else if (nChar == 'D')
      {
            *eye += D3DXVECTOR3(0.0f, 0.0f, 1.0f) * delta;
            *lookAt += D3DXVECTOR3(0.0f, 0.0f, 1.0f) * delta;
      }
      D3DXMatrixLookAtLH( &matView, eye, lookAt, viewUp);
}
```

Funkcija `OnDestroyDevice` služi za oslobađanje zauzetih resursa.

```
void CALLBACK OnDestroyDevice( void* pUserContext )
{
	SAFE_DELETE(configParser);

	SAFE_DELETE(eye);
	SAFE_DELETE(lookAt);
	SAFE_DELETE(viewUp);

	SAFE_DELETE_ARRAY(pMesh);
	SAFE_DELETE_ARRAY(textures);

	SAFE_RELEASE(font);

	ReleaseSPEWorld(pWorld);
}
```

Sada kada smo naveli osnovne funkcije koje će DXUT pozivati, vrijeme je da spomenemo SPE funkcije, klase koje nam omogućuju parsiranje osnovnih postavki (`ConfigParser`) i scene (`SceneParser`), te klasa koja predstavlja objekt u sceni (`SceneObject`).

Kako bi mogli pohranjivati objekte SPE svijeta, deklariramo globalno polje:

```
SPEArray<LPSPERIGIDBODY> Bodies;
```

Te naravno trebamo i pokazivač na SPE svijet:

```
LPSPEWORLD  pWorld;
```

Inicijalizacija i postavljanje SPE svijeta je već opisano ranije kod funkcije `OnCreateDevice`. Funkcija koja provjerava lomljenje objekata se zove `CheckBreak`, i već je ranije spomenuta pri opisu funkcije `OnFrameMove`, pošto ju ona poziva. Funkcija `CheckBreak` se sastoji od 3 dijela. Prvi dio provjerava da li ima tijela koja se lome. Ukoliko ima, funkcija se nastavlja, inače se prekida.

```
if (pWorld->GetBreakList() == 0) return;
```

Drugi dio funkcije se brine oko lomljenja objekata. Računa se ravnina koja će rezati objekt, te kada nastaju novi objekti dodjeljujemo im podatke (`UserData`), točnije teksture i karakteristike prvobitnog objekta. Ukoliko novi objekt ima volumen manji od minimalnog, objekt se ne stavlja u scenu. Isto tako ukoliko novi objekt ima volumen manji od minimalnog za lomljenje, objekt se ne postavlja kao kandidat za daljnje lomljenje.

```
for (int i = 0; i < pWorld->List.size; i++)
{

	LPSPERIGIDBODY pBreakBody = pWorld->List[i];
	LPSPECONTACT pContact = pBreakBody->GetBreakContact();
	srand(timeGetTime());
	SPEVector n(rnd(1), rnd(1), rnd(1));
	SPEPlane pl(n, pContact->GetVirtualConstraint()->GetPosition());
	pWorld->Carve(pBreakBody, pl);
	LPSPERIGIDBODY pbody;

	for (int j = 0; j < pWorld->Meshes.size; j++)
	{
		pWorld->Shape->Initialize(pWorld->Meshes[j]);

		ObjectUserData* objUserData =
			((ObjectUserData*)pBreakBody->UserData)->GetCopy();

		if (pWorld->Shape->GetVolume()  <
			objUserData->minBodyVolume) continue;

		pbody = pWorld->AddRigidBody();

		if (pWorld->Shape->GetVolume() >
			objUserData->minBreakableBodyVolume)
		{
			pbody->SetBreakForce(objUserData->breakForce);
		}

		pbody->PatternState(pBreakBody);

		pbody->UserData = objUserData;

		Bodies.push(pbody);
	}

	pBreakBody->iUserData = 1;
}
```

Treći i ujedno zadnji dio funkcije se brine da izbrišemo objekte koji su rezani, odnosno prvobitni objekti. Njih smo označili sa zastavicom (`iUserData`), te ukoliko je ta zastavica postavljena objekt brišemo.

```
for (int i = 0; i < Bodies.size; i++)
{
	if (Bodies[i]->iUserData == 1)
	{
		SAFE_DELETE(Bodies[i]->UserData);
		pWorld->DeleteRigidBody(Bodies[i]);
		Bodies.qdel(i);
		i--;
	}
}
```

Scena i objekt scene

Nakon što smo objasnili glavne dijelove aplikacije koji se brinu o iscrtavanju i lomljenju objekata, vrijeme je da kažemo nešto o samim objektima, te kako ih unosimo u scenu. U prostoru imena `Scene` možemo naći dvije poveće klase: `SceneObject` i `SceneParser`. Klase komuniciraju na način da objekt tipa `SceneParser` parsira datoteku u koju je spremljena scena, te stvara objekte tipa `SceneObject`. Definirali smo objekta tako da može poprimiti slijedeće parametre:

- početna pozicija
- početna brzina
- početna kutna brzina
- skaliranje u odnosu na mrežu poligona
- masa objekta
- sila koja je potrebna da bi se objekt lomio
- mreža poligona
- unutarnja tekstura
- vanjska tekstura
- da li je tijelo statično
- najmanji mogući volumen tijela
- najmanji mogući volumen lomljenog tijela

Spomenute smo parametre kroz prethodna poglavlja već objasnili. Ovdje bi naime mogli spomenuti zašto baš svaki objekt se sastoji od svih tih parametara? Odgovor je poprilično jednostavan: „Želimo imati mogućnost staviti u scenu što specifičnije objekte.“ Pa tako početna pozicija, brzina, kutna brzina i masa (gravitacija) definiraju kretanje objekta. Ukoliko je tijelo statično uopće ne postoji kretanje. Skaliranje i poligonalna mreža objekta definiraju oblik objekta. Teksture definiraju izgled objekta. Te ono najzanimljivije

najmanji volumen tijela i lomljenog tijela sa silom lomljenja definiraju kako će se tijelo lomiti. Članske varijable koje sadržavaju spomenute parametre su:

```
float position[3];
float velocity[3];
float angularVelocity[3];
float scaling[3];
float mass;
float breakForce;
int meshFilenameId;
int insideTextureFilenameId;
int outsideTextureFilenameId;
bool isStatic;
float minBodyVolume;
float minBreakableBodyVolume;
```

Na prvi pogled je možda malo zbunjujuće što to točno radimo sa imenom datoteke koja sadrži mrežu poligona i imenima datoteka tekstura. Kako scena može sadržavati poveći broj objekata sa istom poligonalnim mrežama i/ili istim teksturama, bilo bi neodgovorno za svaki objekt pozivati parsiranje spomenutih datoteka. Zato smo uveli mogućnost definiranja tih datoteka, te tako objektu samo dodjeljujemo broj definicije. Primjer scene sa dva objekta (od toga jedan statičan, dok drugi nije) je:

```
defineMeshFilename 0 : media/box.x
defineTextureFilename 0 : media/dark_brick.jpg
defineTextureFilename 1 : media/brickwall01.jpg
defineTextureFilename 2 : media/whitefloor.jpg

{
position : 0 0 0
scaling : 20 0.5 20
mass : 10000
useMeshFilename : 0
useInsideTextureFilename : 2
useOutsideTextureFilename : 2
static : true
}
{
position : 0 50 0
velocity : 0 -100 0
angularVelocity : 0 0 0
scaling : 1 1 1
mass : 3000
breakForce : 10000
useMeshFilename : 0
useInsideTextureFilename : 0
useOutsideTextureFilename : 1
static : false
minBodyVolume : 0.0009
minBreakableBodyVolume : 0.00000005
}
```

Sintaksa za definiciju mreže poligona je:

```
defineMeshFilename <CJELOBROJNI IDENTIFIKATOR> : <IME DATOTEKE>
```

Sintaksa za definiciju teksture je:

```
defineTextureFilename <CJELOBROJNI IDENTIFIKATOR> : <IME DATOTEKE>
```

Pošto objekt ima popriličan broj parametara definirali smo početak definicije objekta „{“ i kraj definicije objekta „}“.

Sintaksa za definiciju početne pozicije objekta:

```
position : <X KOORDINATA> <Y KOORDINATA> <Z KOORDINATA>
```

Sintaksa za definiciju početne brzine objekta:

```
velocity : <X KOORDINATA> <Y KOORDINATA> <Z KOORDINATA>
```

Sintaksa za definiciju početne kutne brzine objekta:

```
angularVelocity : <X KOORDINATA> <Y KOORDINATA> <Z KOORDINATA>
```

Sintaksa za definiciju skaliranja objekta:

```
scaling : <X KOORDINATA> <Y KOORDINATA> <Z KOORDINATA>
```

Sintaksa za definiciju mase objekta:

```
mass : <MASA OBJEKTA>
```

Sintaksa za definiciju sile lomljenja objekta:

```
breakForce : <SILA LOMLJENJA OBJEKTA>
```

Sintaksa za definiciju mreže poligona objekta:

```
useMeshFilename : <CJELOBROJNI IDENTIFIKATOR>
```

Sintaksa za definiciju unutarnje teksture objekta:

```
useInsideTextureFilename : <CJELOBROJNI IDENTIFIKATOR>
```

Sintaksa za definiciju unutarnje teksture objekta:

```
useOutsideTextureFilename : <CJELOBROJNI IDENTIFIKATOR>
```

Sintaksa za definiciju statičnosti objekta:

```
static : (true|false)
```

Sintaksa za definiciju minimalnog volumena objekta:

```
minBodyVolume : <MINIMALNI VOLUMEN>
```

Sintaksa za definiciju minimalnog volumena lomljenog objekta:

```
minBreakableBodyVolume : <MINIMALNI VOLUMEN>
```

Svi nezavršni znakovi u navedenoj sintaksi su realni brojevi, osim ako nije drugačije spomenuto. Iza svake definicije potrebno je staviti oznaku kraja reda kako bi parser uspješno parsirao datoteku.

Na početku aplikacije poziva se `SceneParser` koji parsira, te stvara objekte scene tipa `SceneObject`. Stvoreni objekti se zatim postavljaju u SPE svijet, s ekvivalentnim SPE tipovima, te tako više `SceneObject` nije u potpunosti potreban. Kada kažemo nije u potpunosti potreban zapravo želimo reći da nam nisu svi parametri objekta više potrebni. Neki s druge strane jesu, kao na primjer teksture pošto nam trebaju za iscrtavanje. Kako bi oslobodili memoriju od `SceneObject` objekata, a s druge strane zadržali parametre objekta koji su nam potrebni koristimo klasu `ObjectUserData` koja sadrži slijedeće članske varijable:

```
LPD3DXMESH pMesh;
int insideTextureId;
int outsideTextureId;
float minBodyVolume;
float minBreakableBodyVolume;
float breakForce;
```

Teksture su nam potrebne kako smo već spomenuli za iscrtavanje, dok je mreža poligona, najmanji volumeni i sila lomljenja potrebni kako bi ispravno lomili objekt.

Na posljetku možemo spomenuti metodu `SetObjects` koja postavlja objekte u scenu. Metoda vješto barata parserom scene `SceneParser`, te tako dohvaća scenu. Metodu možemo podijeliti u nekoliko koraka:

Dohvaćanje parsera scene:

```
sceneParser = new SceneParser();
      if (sceneParser->StartParsing(
            configParser->GetSceneObjectsFilename()) == ERROR)
      {
            exit(0);
      }
```

Dohvaćanje mreža poligona koji se koriste u sceni:

```
int nMesh = sceneParser->GetNumberOfMeshes();
      pMesh = new LPD3DXMESH[nMesh];
      Definition* meshDefinitions = new Definition[nMesh];
      Definition* meshDef;

      for (int i = 0; i < nMesh; i++)
      {
            meshDef = sceneParser->GetMesh();
            meshDefinitions[meshDef->GetId()] = *meshDef;
      }
```

Dohvaćanja tekstura koje se koriste u sceni:

```
int nTexture = sceneParser->GetNumberOfTextures();
      textures = new LPDIRECT3DTEXTURE9[nTexture];
      Definition* texDef;
      sceneParser->ResetParser();

      for (int i = 0; i < nTexture; i++)
      {
            texDef = sceneParser->GetTexture();
            D3DXCreateTextureFromFile(pd3dDevice,
                  A2W(texDef->GetFilename()),
                  &textures[texDef->GetId()]);
      }
```

Dohvaćanje i postavljanje statičnih objekata scene:

```
D3DXMatrixScaling(&mat, sceneObject->GetScalingX(),
                        sceneObject->GetScalingY(),
                        sceneObject->GetScalingZ());

LoadMesh(pd3dDevice, pMesh[sceneObject->GetMeshFilenameId()],
      A2W(meshDefinitions[
            sceneObject->GetMeshFilenameId()].GetFilename()),
       mat);
InitShape(pShape, pMesh[sceneObject->GetMeshFilenameId()]);
pbody = pWorld->AddRigidBody(pShape);
pbody->SetBeStatic(true);
pbody->SetMass(sceneObject->GetMass());
pbody->SetPosition(SPEVector(sceneObject->GetPositionX(),
                             sceneObject->GetPositionY(),
                             sceneObject->GetPositionZ()));

ObjectUserData* userData = new ObjectUserData();
userData->pMesh = pMesh[sceneObject->GetMeshFilenameId()];
userData->insideTextureId = sceneObject->GetInsideTextureFilenameId();
userData->outsideTextureId = sceneObject->GetOutsideTextureFilenameId();
pbody->UserData = userData;
Bodies.push(pbody);
```

Dohvaćanje i postavljanje dinamičkih objekata scene:

```
D3DXMatrixScaling(&mat, sceneObject->GetScalingX(),
                        sceneObject->GetScalingY(),
                        sceneObject->GetScalingZ());

LoadMesh(pd3dDevice, pMesh[sceneObject->GetMeshFilenameId()],
      A2W(meshDefinitions[
            sceneObject->GetMeshFilenameId()].GetFilename()),
      mat);
InitMesh(speMesh, pMesh[sceneObject->GetMeshFilenameId()]);
pShape->Initialize(speMesh);
pbody=pWorld->AddRigidBody(pShape);

pbody->SetPositionMesh(SPEVector(sceneObject->GetPositionX(),
                                 sceneObject->GetPositionY(),
                                 sceneObject->GetPositionZ()));

pbody->SetAngularVelocity(SPEVector(sceneObject->GetAngularVelocityX(),
                                    sceneObject->GetAngularVelocityY(),
                                    sceneObject->GetAngularVelocityZ()));

pbody->SetVelocity(SPEVector(sceneObject->GetVelocityX(),
                             sceneObject->GetVelocityY(),
                             sceneObject->GetVelocityZ()));

pbody->SetMass(sceneObject->GetMass());

pbody->SetBreakForce(sceneObject->GetBreakForce());

ObjectUserData* userData = new ObjectUserData();
userData->pMesh = NULL;
userData->insideTextureId = sceneObject->GetInsideTextureFilenameId();
userData->outsideTextureId = sceneObject->GetOutsideTextureFilenameId();
userData->minBodyVolume = sceneObject->GetMinBodyVolume();
userData->minBreakableBodyVolume =
                               sceneObject->GetMinBreakableBodyVolume();
userData->breakForce = sceneObject->GetBreakForce();
pbody->UserData = userData;
Bodies.push(pbody);
```

Te na kraju oslobađanje memorije:

```
pWorld->ReleaseShape(pShape);

sceneParser->EndParsing();

SAFE_DELETE_ARRAY(meshDefinitions);
SAFE_DELETE(sceneParser);
```

Postavke aplikacije

Kako bi mogli jednostavno izmjenjivati postavke aplikacije bez ponovnog prevođenja kôda koristimo klasu `ConfigParser`. Klasa parsira datoteku `config.txt` koja sadrži slijedeće parametre:

- širina ekrana
- visina ekrana
- očište
- gledište
- ime datoteke sa scenom
- način iscrtavanja (normalno ili žičano)

Primjer jedne postavke aplikacije, odnosno jedne datoteke `config.txt` može biti:

```
width : 1680
height : 1050
eyePosition : 10 1 0
lookAtPosition : 0 0 0
sceneObjectsFilename : scene_objects10.txt
wireframe : false
```

Sintaksa za definiciju širine ekrana:

```
width : <CJELOBROJNA_ŠIRINA_EKRANA>
```

Sintaksa za definiciju visine ekrana:

```
height : <CJELOBROJNA_VISINA_EKRANA>
```

Sintaksa za definiciju očišta:

```
eyePosition : <X KOORDINATA> <Y KOORDINATA> <Z KOORDINATA>
```

Sintaksa za definiciju gledišta:

```
lookAtPosition : <X KOORDINATA> <Y KOORDINATA> <Z KOORDINATA>
```

Sintaksa za definiciju datoteke scene:

```
sceneObjectsFilename : <IME DATOTEKE>
```

Sintaksa za definiciju iscrtavanja:

```
wireframe : (true|false)
```

Ukoliko želimo normalno iscrtavanje ovaj parametar postavljamo na `false`, inače za žičano iscrtavanje postavljamo na `true`.

Pravila parsiranja su jednaka kao i kod prethodnog parsera `SceneParser`.

Naslov: Simulacija lomljenja čvrstih tijela

Sažetak: Ovaj rad obrađuje simulaciju lomljenja čvrstih tijela u stvarnom vremenu. Prezentiran je osnovni fizikalni model kretanja čvrstih tijela, te tri načina implementacije lomljenja čvrstih tijela na gotovim sustavima za simulaciju fizike u stvarnom vremenu. Prvi način implementacije lomljenja tijela temelji se na dijeljenju objekata zadanom ravninom. Drugi način implementacije koristi metodu konačnih elemenata i zglobove sustava za simulaciju fizike za međusobno povezivanje elemenata. Dok treći način implementacije koristi metodu konačnih elemenata i sustav za simulaciju fizike sa podrškom za simulaciju trenja. Uz rad su priloženi izvorni kôdovi implementacija uporabom sva tri načina lomljenja tijela.

Ključne riječi: model fizike čvrstog tijela, simulacija u stvarnom vremenu, lomljenje, pucanje, metoda konačnih elemenata, sustavi za simulaciju fizike

Title: Simulation of breaking rigid bodies

Abstract: In this paper we work on real-time simulation of breaking rigid bodies. We present an introduction to physically based modeling of rigid body dynamics and three methods how to implement breaking of rigid bodies on physics engines in real-time. First method of implementation is based on dividing objects by plane. The second method of implementation is using finite element method and joints of physics engine. While third method of implementation is using finite element method and physics engine which supports friction simulation. Source codes of implementations using all three methods are attached with this paper.

Keywords: physically based modeling, real-time simulation, fracture, cracking, finite element method, physics engine