Simulacija lomljivih objekata u stvarnom vremenu

Marković-Đurin, Luka

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:520267

Rights / Prava: In copyright/Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-14**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 459

SIMULACIJA LOMLJIVIH OBJEKATA U STVARNOM VREMENU

Luka Marković-Đurin

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 459

SIMULACIJA LOMLJIVIH OBJEKATA U STVARNOM VREMENU

Luka Marković-Đurin

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 459

Pristupnik:	**Luka Marković-Đurin (0036524734)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Simulacija lomljivih objekata u stvarnom vremenu**

Opis zadatka:

Proučiti mehanizme lomljenja objekata i stvaranje fraktura i krhotina. Razmotriti mogućnosti implementacije vizualizacije i simulacije lomljenja uz dinamičko ostvarivanje razaranja objekata. Posebice obratiti pažnju na mehanizme ostvarive u stvarnom vremenu. Implementirati proučene mehanizme te ostvariti vizualizaciju simulacije razaranja i lomljenja objekata. Načiniti testiranje na nizu primjera. Analizirati i ocijeniti ostvarene rezultate. Diskutirati upotrebljivost ostvarenih rezultata kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Koristiti grafički programski pogon Unity i programski jezik C#. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 28. lipnja 2024.

Sadržaj

Uvod 1

1. Primjena teksture oštećenja 2

2. Zamjena oštećenim modelom 4

 2.1. Zamjena fragmentiranim modelom 5

3. Fragmentacija objekta u stvarnom vremenu 7

 3.1. Rezanje objekta 7

  3.1.1. Rezanje trokuta 8

 3.2. Triangulacija 9

  3.2.1. Ubrzanje algoritma 13

 3.3. Ograničena triangulacija 14

 3.4. Fragmentacija 17

 3.5. Razdvajanje otoka 19

 3.6. Konveksna dekompozicija 20

4. Uzorak prijeloma 21

5. Simulacija fizike fragmenata 24

6. Rezultati 25

Zaključak 28

Literatura 29

Sažetak 30

Summary 31

Skraćenice 32

Privitak 33

Uvod

Simulacija lomljivih objekata podrazumijeva simulaciju pojava poput razaranja okoliša, lomljenja stakla ili uništavanja objekata u virtualnoj sceni.

Potreba za simulacijom lomljivih objekata u stvarnom vremenu javlja se u interaktivnim virtualnim svjetovima poput računalnih i videoigara. Od najstarijih igara poput *Space Invaders* i *Worms* do modernih visokobudžetnih naslova poput *Battlefield* igara (Slika 0.1 Simulacija urušavanja kuće u računalnoj igri *Battlefield: Bad Company 2*), interakcija igrača i okoliša kroz simulaciju lomljivih objekata igra ključnu ulogu u ostvarivanju karakteristika igre te uranjanja igrača u virtualni svijet.

Zbog zahtjeva za izvođenjem u stvarnom vremenu implementacije simulacije lomljivih objekata moraju balansirati između realističnosti simulacije te brzine i zahtjevnosti izvođenja. Ključan kriterij je stoga često vizualna privlačnost simulacije umjesto znanstveno utemeljenih metoda. Zahtjevi simulacije intrinzično su povezani i s kontekstom igre odnosno virtualnog svijeta u kojem se implementira. Natjecateljske igre poput *Counter Strike 2* ili *Valorant* simulaciju lomljivih objekata svode na jednostavna površinska oštećenja u svrhu smanjenja vizualne buke te poboljšanja performansi igre. S druge strane visokobudžetne igre s naglaskom na realizam zahtijevaju mnogo kompleksnija rješenja za postizanje vizualno prihvatljivih rezultata.

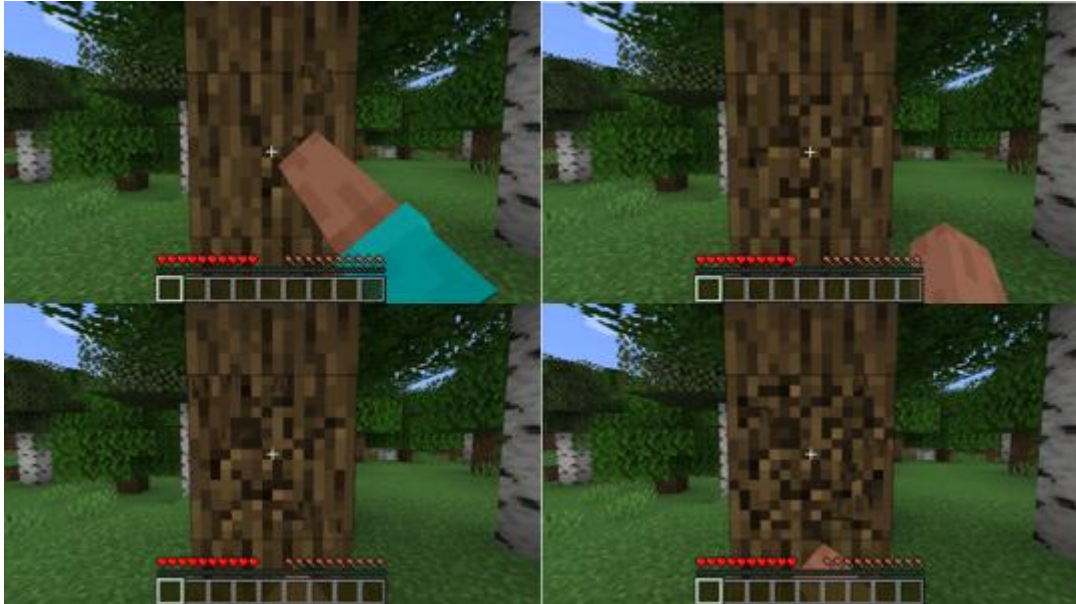
Ovaj rad daje pregled različitih metoda simulacije lomljivih objekata u stvarnom vremenu te analizira njihove prednosti, nedostatke te prikladne primjere uporabe. Za određene pristupe napravljena je te opisana njihova implementacija u pogonskom sustavu *Unity*.

Slika 0.1 Simulacija urušavanja kuće u računalnoj igri *Battlefield: Bad Company 2*

1. Primjena teksture oštećenja

Primjena teksture oštećenja predstavlja pristup u kojem se objektu primjenjuje dodatna ili mijenja trenutna tekstura kako bi se simuliralo oštećenje objekta. Ovaj pristup vrlo je popularan za simulaciju površinski oštećenja poput rupe od metka (Slika 1.1 Primjena teksture rupe od metka za prikaz površinskog oštećenja u računalnoj igri *Counter Strike 2*) ili za prikaz različitih razina oštećenja objekta (Slika 1.2 Primjena teksture za prikaz različitih razina oštećenja bloka u računalnoj igri *Minecraft*).

Slika 1.1 Primjena teksture rupe od metka za prikaz površinskog oštećenja u računalnoj igri *Counter Strike 2*



Slika 1.2 Primjena teksture za prikaz različitih razina oštećenja bloka u računalnoj igri *Minecraft*

Kao poboljšanje ove metode, teksturi oštećenja često se pridodaje tekstura normala kako bi se tekstura prilagođavala okolnom osvjetljenju objekta. Ovaj pristup često se koristi u kombinaciji s sustavom čestica za prikaz prašine i krhotina nastalih pri oštećenju objekta.

Jedna od prednosti ovog pristupa je u jednostavnosti implementacije. Mnogi programski pogoni poput *Unreal Engine*, *Unity* te *Godot* omogućavaju lako mijenjanje teksture tijekom izvođenja programa što olakšava implementaciju ove metode umjetnicima koji rade na izradi igara ili virtualnih svjetova. Nadalje nezavisnost ove metode o ostalim objektima u sceni omogućuje širok stupanj primjenjivosti. Tekstura oštećenja često ne ovisi o objektu na koji se primjenjuje te ju je stoga moguće primijeniti na mnoštvo objekata u sceni bez potrebe za nadogradnjom implementacije. Implementacija ovog pristupa omogućava prikaz lokalnih oštećenja zbog mogućnosti primjene teksture oštećenja na potreban dio objekta. Ovaj pristup također nije vrlo računski zahtjevan budući da uključuje samo izmjenu teksture objekta.

Glavni nedostatak ovog pristupa je ograničenost u njegovoj primjenjivosti. Kao što je već navedeno ovaj pristup prikladan je za prikaz površinskih oštećenja objekata te ne omogućuje prikaz strukturalnih oštećenja, odnosno izmjenu samog modela objekta i simulaciju nastalih fragmenata.

2. Zamjena oštećenim modelom

Implementacija ovog pristupa uključuje pripremu nekoliko verzija modela objekta koje odgovaraju različitim razinama oštećenja objekta. Model objekta se tijekom izvođenja programa dinamički zamjenjuje oštećenim modelom ovisno o razini oštećenja.

Slika 2.1 Zamjena oštećenih modela vrata u računalnoj igri *Counter Strike: Global Offensive* [29]

Prednost ovog pristupa je jednostavnost implementacije. Kao i pri primjeni oštećene strukture, programski pogoni za razvoj igara omogućavaju laku izmjenu modela objekta tijekom izvođenja programa. Kao dodatno ubrzanje, sve modele moguće je odmah učitati pri pokretanju programa ili učitavanju scene te je za izmjenu modela potrebno isključiti prikaz starog te omogućiti prikaz novog modela. Ova implementacija nije vrlo zahtjevna te omogućuje prikaz strukturalnog oštećenja modela objekta.

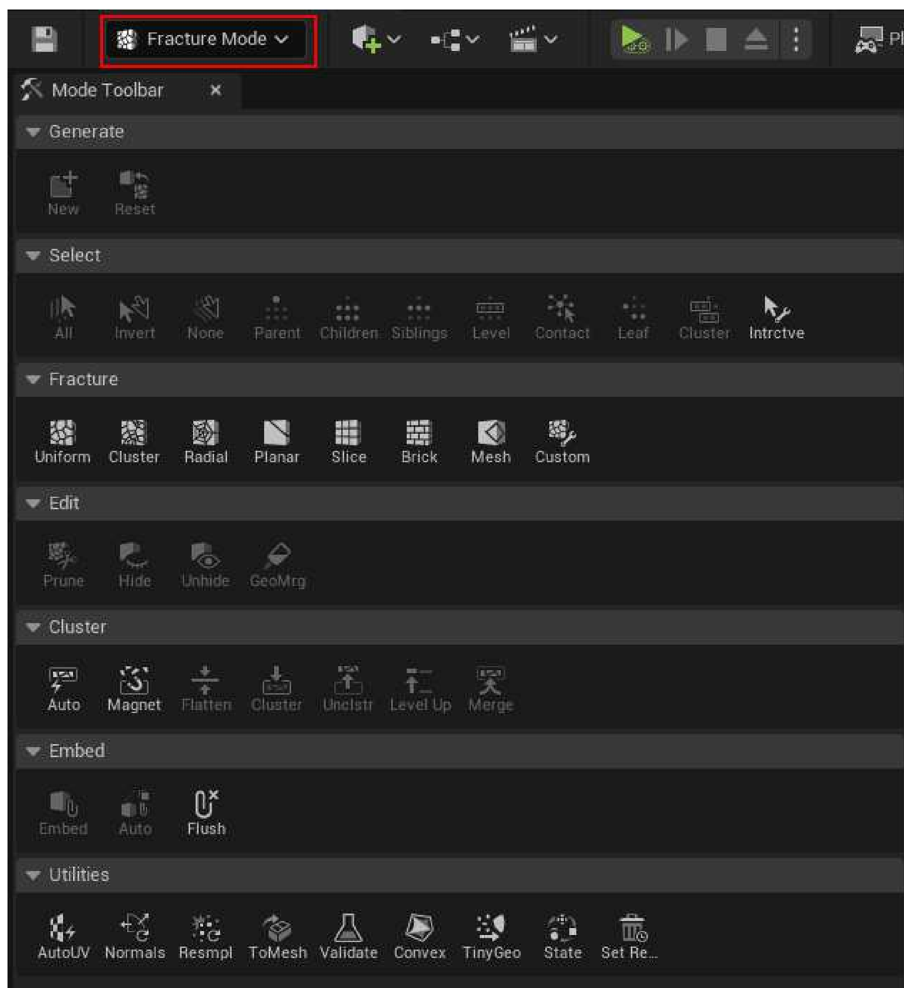
Glavni nedostatak ovog pristupa je potreba za pripremom više modela za isti objekt što dovodi do produljenog vremena razvoja te povećanih troškova. Većina modela koje je moguće preuzeti na internetu nisu pripremljeni za ovu namjenu te je iz njih stoga potrebno stvoriti dodatne oštećene modele. Dodatni nedostatak je nemogućnost prikaza lokalnih oštećenja. Za N točaka mogućih oštećenja bilo bi potrebno pripremiti 2^N različitih modela što je prezahtjevno za gotovo sve implementacije. Umjesto toga implementacije poput oštećenja vrata u računalnoj igri *Counter Strike: Global Offensive* (Slika 2.1 Zamjena oštećenih modela vrata u računalnoj igri *Counter Strike: Global Offensive*) uvijek rezultiraju jednakim slijedom modela neovisno o točki nastanka oštećenja.

Ovaj pristup uglavnom se primjenjuje kada je zahtjev za realističnosti relaksiran u korist smanjenja zahtjeva za računalnim resursima te bržem izvođenju programa. Predvidljivost nastanka oštećenja objekta također može biti privlačna pri implementaciji natjecateljskih igara.

2.1. Zamjena fragmentiranim modelom

Zamjena fragmentiranim modelom predstavlja podskup metode zamjene oštećenim modelom, u kojem se objekt zamjenjuje nizom fragmenata nastalih na temelju originalnog modela objekta.

Ovaj pristup vrlo je popularan zato što omogućava simulaciju fragmenata bez potrebe za fragmentiranjem tijekom izvođenja programa što smanjuje zahtjev za računalnim resursima. Popularnost ovog pristupa pri implementaciji igara očituje se u tome da pogonski sustav *Unreal Engine* posjeduje gotovo rješenje za fragmentiranje objekata u osnovnoj verziji programa (Slika 2.2 Fracture Mode alat za fragmentiranje objekata unutar urednika programskog pogona Unreal Engine 5). Osim *Unreal Engine*-a popularna rješenja za fragmentiranje objekata uključuju *Cell Fracture* programa *Blender* te *Fracture Fx* programa *Maya*.



Slika 2.2 Fracture Mode alat za fragmentiranje objekata unutar urednika programskog pogona Unreal Engine 5

Nedostaci ovog pristupa uključuju potrebu za dodatnom pripremom objekata (fragmentiranje) te nemogućnost simulacije lokalnih oštećenja. Budući da objekt fragmentiramo prije samog pokretanja programa stvaranje fragmenata pri simulaciji lomljenja objekta je uvijek isto što može biti prednost ili nedostatak ovisno o kontekstu uporabe.

Prednosti ovog pristupa su mogućnost korištenja gotovih alata za postizanje vizualno privlačnih simulacija lomljenja odnosno uništavanja objekata i okoliša. Mogućnost uparivanja gotovog sustava za fragmentaciju objekta te gotovih sustava za simulaciju fizike alata za razvoj igara omogućava impresivne simulacije razaranja što je i dokazano u demonstracijskoj VR igri *Robo Recall* napravljenoj u sustavu *Unreal Engine* (Slika 2.3 Snimka zaslona iz igre *Robo Recall* napravljene za demonstraciju alata *Chaos Destruction* pogonskog sustava *Unreal Engine*).

Slika 2.3 Snimka zaslona iz igre *Robo Recall* napravljene za demonstraciju alata *Chaos Destruction* pogonskog sustava *Unreal Engine*

Primjenjivost ovog pristupa posebno se očituje pri potrebi simulacije uništavanja vrlo kompleksnih modela s velikim brojem poligona, budući da bi njihovo fragmentiranje tijekom izvođenja programa bilo prezahtjevno za simulaciju u stvarnom vremenu.

3. Fragmentacija objekta u stvarnom vremenu

Fragmentacija objekta u stvarnom vremenu omogućava vrlo fleksibilan pristup simulaciji lomljivih objekata. Budući da se fragmentacija izvodi tijekom izvođenja programa omogućava nam da dinamički prilagodimo simulaciju različitim uvjetima. Također budući da je fragmentacija implementirana u kodu, može se primijeniti na gotovo sve objekte u sceni bez potrebe za dodatnim pripremama kao u ranije opisanim pristupima. Zbog zahtjeva za izvođenjem u stvarnom vremenu, ovaj pristup je često neprikladan za simulaciju nad vrlo kompleksnim modelima s velikim brojem poligona.

Ovaj odjeljak opisuje implementaciju fragmentacije u stvarnom vremenu te moguća poboljšanja i proširenja implementacije.

3.1. Rezanje objekta

Prvi korak implementacije fragmentacije je implementacija rezanja objekta. Rezanjem objekta nastaju dva fragmenta, nazovimo ih gornji te donji fragment. Za svaki vrh originalnog modela potrebno je odrediti pripada li gornjem ili donjem fragmentu. Opišemo li rez ravninom definiranom točkom i normalom, provjeru je li točka iznad ili ispod ravnine možemo napraviti formulom (1).

$$d = \left(\overrightarrow{P_x} - \overrightarrow{P_0}\right) \cdot \vec{n} \tag{1}$$

Gdje je P_x točka za koju radimo provjeru, P_0 točka ravnine, a n normalizirana normala ravnine.
Razlikujemo tri slučaja:

1. $d > 0$: točka je iznad ravnine
2. $d = 0$: točka je na ravnini
3. $d < 0$: točka je ispod ravnine

Točke koje se nalaze iznad te na ravnini pridjeljujemo gornjem fragmentu, a točke ispod ravnine donjem fragmentu.

Nakon razvrstavanja točki u pripadajuće fragmente, potrebno je odrediti pripadnost trokuta pojedinom fragmentu.

Razlikujemo tri slučaja:

1. Sve tri točke trokuta su iznad ravnine reza: trokut pripada gornjem fragmentu
2. Sve tri točke trokuta su ispod ravnine reza: trokut pripada donjem fragmentu
3. Jedna/dvije točke su iznad, a preostale točke/točka ispod ravnine reza: potrebno je prerezati trokut

3.1.1. Rezanje trokuta

Kako bi presjekli trokut, potrebno je pronaći točke presjeka između stranica trokuta te ravnine reza. Točke presjeka nalazimo na stranicama trokuta između parova vrhova trokuta koji pripadaju različitim fragmentima. Točku presjeka dužine te ravnine možemo pronaći formulom (2).

$$\vec{P} = \overrightarrow{P_1} + (\overrightarrow{P_2} - \overrightarrow{P_1}) \cdot s \tag{2}$$

Gdje su P_1 i P_2 točke dužine, a s parametar dužine izračunat formulom (3).

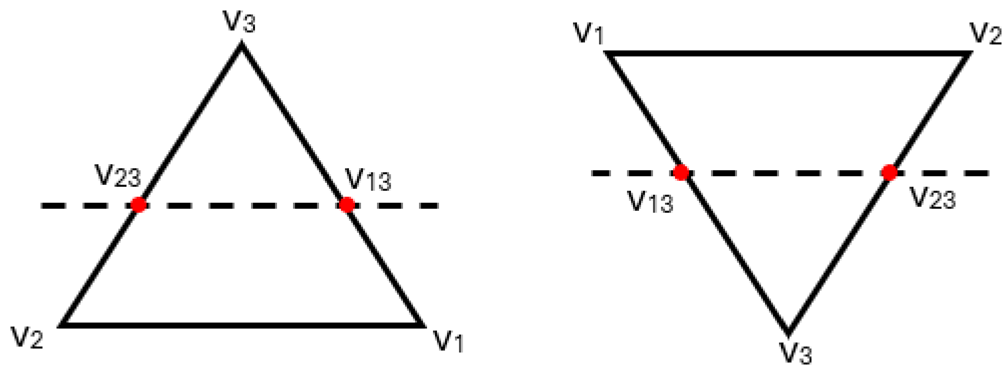
$$s = \frac{(\overrightarrow{P_0} - \overrightarrow{P_1}) \cdot \vec{n}}{(\overrightarrow{P_2} - \overrightarrow{P_1}) \cdot \vec{n}} \tag{3}$$

Gdje je P_0 točka, a n normala ravnine. Ukoliko je parametar s izvan ograničenja [0, 1], ne postoji presjek između dužine i ravnine.

Presjekom trokuta nastaju dva segmenta od kojih je jedan trokut a drugi četverokut. Ovisno o orijentaciji trokuta razlikujemo dva slučaja:

1. Jedan vrh iznad, dva vrha ispod ravnine reza: iznad ravnine je trokut, ispod četverokut
2. Dva vrha iznad, jedan ispod ravnine reza: iznad ravnine je četverokut, ispod trokut

Prikaz ovih slučajeva vidljiv je na slici Slika 3.1 Presjek trokuta ovisno o orijentaciji.



Slika 3.1 Presjek trokuta ovisno o orijentaciji

Točke presjeka (V_{13}, V_{23}) dodajemo u oba segmenta. Pri slučaju 1, gornjem fragmentu dodajemo trokut $V_{13}V_{23}V_3$, a donjem fragmentu trokute $V_1V_2V_{13}$ te $V_2V_{23}V_{13}$. Istovjetno činimo i pri slučaju 2 gdje gornjem fragmentu dodajemo dva, a donjem jedan trokut.

Parametar *s* izračunat u formuli (3), možemo iskoristiti za izračun normala te koordinata teksture u novonastalim vrhovima V_{13} te V_{23}. Izračun vršimo linearnom interpolacijom kao u formuli (4). Interpolirane normale je potrebno dodatno normalizirati.

$$\overrightarrow{n_{13}} = \overrightarrow{n_1} + s_{13}(\overrightarrow{n_3} - \overrightarrow{n_1}) \tag{4}$$

3.2. Triangulacija

Budući da je model objekta predstavljen mrežom trokuta, fragmenti nastali nakon reza sadržavat će dio koji nije ispunjen trokutima. Kako bi potrebni dio ispunili trokutima, potrebno je odraditi proces triangulacije nad točkama duž reza. Budući da gornji i donji fragment sadržavaju iste točke duž reza, triangulaciju je potrebno odraditi samo jednom te se izračunati trokuti mogu primijeniti na oba fragmenta uz suprotni poredak vrhova i orijentaciju normala.

Delaunay triangulacija za niz točaka generira set trokuta čije opisane kružnice ne sadržavaju niti jednu točku drugog trokuta. Takva triangulacija kao posljedicu nalazi trokute koji maksimiziraju minimalne kutove, odnosno izbjegava stvaranje uskih izduženih trokuta. To svojstvo čini Delaunay triangulaciju vrlo popularnom za primjenu u računalnoj grafici. Pri implementaciji triangulacije korišten je algoritam temeljen na radu *A fast algorithm for constructing Delaunay triangulations in the plane* [2]. Opis algoritma slijedi u nastavku.

Kao prvi korak triangulacije, točke je potrebno projicirati na ravninu triangulacije, odnosno ravninu reza. Vektore baze za projekciju možemo odabrati kao u izrazu (5).

$$\overrightarrow{b_1} = \frac{\overrightarrow{p_1} - \overrightarrow{p_2}}{|\overrightarrow{p_1} - \overrightarrow{p_2}|}, \overrightarrow{b_2} = \frac{\overrightarrow{b_1} \times \vec{n}}{|\overrightarrow{b_1} \times \vec{n}|} \tag{5}$$

Gdje su p_1 i p_2 dvije proizvoljno odabrane točke u triangulaciji, a n normalizirana normala ravnine reza.

Projicirane koordinate točaka možemo izračunati prema izrazu (6).

$$x = \vec{p} \cdot \overrightarrow{b_1}, y = \vec{p} \cdot \overrightarrow{b_2} \tag{6}$$

Za provođenje algoritma potrebno je konstruirati matricu koja za svaki proizvedeni trokut pohranjuje indekse njegovih vrhova te indekse susjednih trokuta. Izgled takve matrice može se vidjeti u Tablica 3.1 Primjer matrice vrhova i susjedstva trokuta.

TROKUT	V1	V2	V3	tE12	tE23	tE31
0	1	4	2	0	2	3
1	2	4	3	1	0	3

Tablica 3.1 Primjer matrice vrhova i susjedstva trokuta

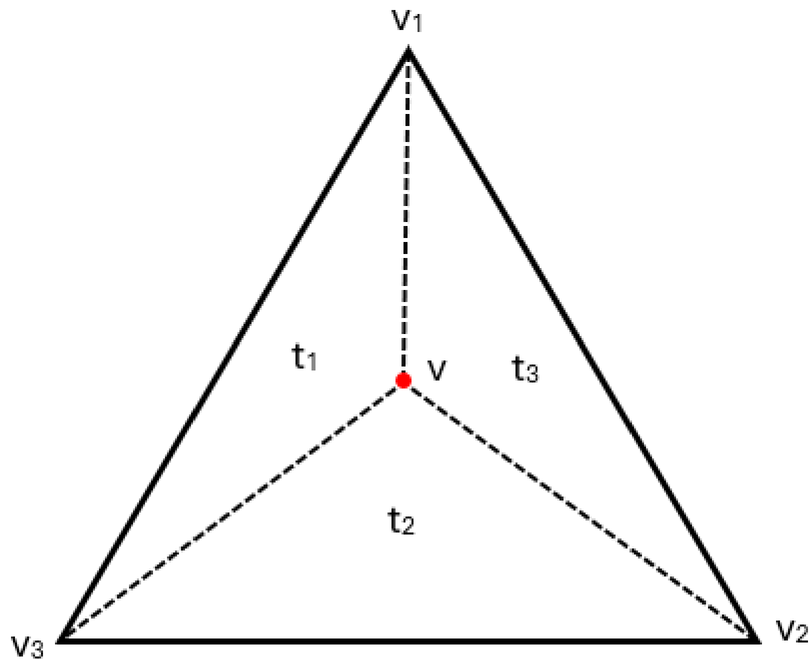
Pri početku algoritma također je potrebno točkama triangulacije dodati tri nove točke. Koordinate točaka su proizvoljne pod uvjetom da tvore trokut koji obuhvaća sve točke u triangulaciji. Podatke o tom trokutu unosimo kao prvi član u matrici vrhova i susjedstva trokuta.

U sljedećem koraku algoritma započinjemo s dodavanjem točaka u triangulaciju. Pri dodavanju svake točke prvo pronalazimo trokut u kojem se ta točka trenutno nalazi. Pronalazak trokuta radimo provjerom nalazi li se ta točka s desne strane sve tri stranice trokuta (poredak vrhova trokuta je u smjeru kazaljke na satu). Provjeru nalazi li se točka s desne strane dužine možemo napraviti izrazom (7).

$$d = (P_{2x} - P_{1x})(P_y - P_{1y}) - (P_{2y} - P_{1y})(P_x - P_{1x}) \quad (7)$$

Gdje su P_1 i P_2 točke dužine, a P točka za koju radimo provjeru. Točka P nalazi se s desne strane dužine ako je vrijednost izraza manja ili jednaka 0. Ako neka od stranica trokuta za kojeg trenutno radimo provjere ne zadovoljava ovaj uvjet, pretragu nastavljamo s trokutom koji je susjedan toj stranici. Ako su uvjeti zadovoljeni za sve tri stranice trokuta, točku dodajemo tom trokutu.

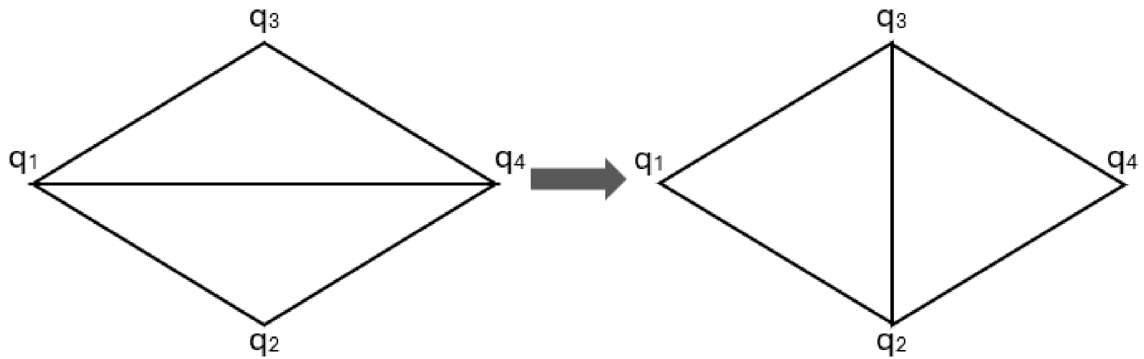
Dodavanje točke trokutu vršimo na način da taj trokut zamjenjujemo s tri nova trokuta formirana iz vrhova starog trokuta te dodane točke. Prikaz ovog koraka nalazi se na slici Slika 3.2 Formiranje novih trokuta pri dodavanju točke. Podatke o novonastalim trokutima dodajemo u matricu vrhova i susjedstva te ažuriramo podatke o susjedstvu susjednim trokutima. Završetkom ovog koraka broj trokuta u triangulaciji povećao se za dva. Budući da smo započeli s početnim trokutom te ovaj korak radimo za svaku točku u triangulaciji, triangulacijom nastaju 2N+1 trokuta, gdje je N broj točaka u triangulaciji.



Slika 3.2 Formiranje novih trokuta pri dodavanju točke

Nakon dodavanja novih trokuta, potrebno je provjeriti zadovoljavaju li uvjet Delaunay triangulacije. Svaki novonastali trokut, te njemu susjedan trokut (koji nije nastao u ovom koraku) dodajemo u LIFO stog. Parovi trokuta tvore četverokute kojima je potrebno zamijeniti dijagonalu ako ne maksimiziraju najmanji kut trokuta. Vizualizaciju ovog koraka

moguće je vidjeti na slici Slika 3.3 Zamjena dijagonala kako bi se zadovoljio uvjet Delaunay triangulacije.

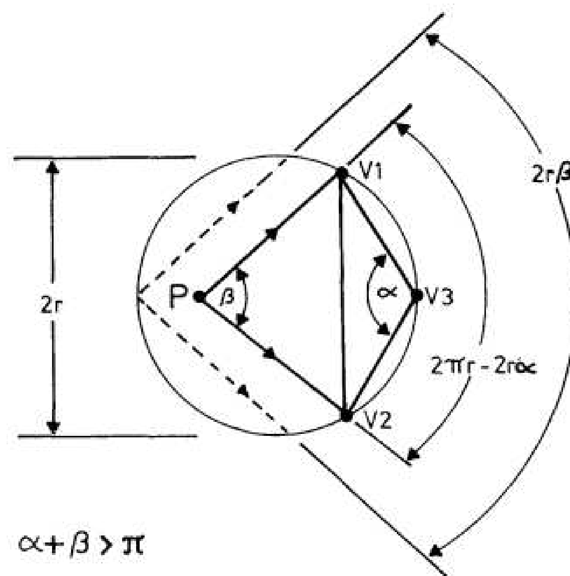


Slika 3.3 Zamjena dijagonala kako bi se zadovoljio uvjet Delaunay triangulacije

Kako bi provjerili zadovoljavaju li trokuti uvjet Delaunay triangulacije potrebno je provjeriti nalazi li se u njihovim opisanim kružnicama točka iz nasuprotnog trokuta.

Neka je α ∠V1V3V2, a β ∠V1PV2. Točka P nalazi se unutar trokuta △V1V2V3, ako i samo ako je kružni luk kuta α manji od kružnog luka kuta 2β. [29] Ovaj uvjet ekvivalentan je izrazu (8). Vizualizacija ovog izraza nalazi se na slici Slika 3.4 Geometrijski uvjet za pripadnost točke P opisanoj kružnici [29].

$$2\pi r - 2r\alpha < 2r\beta$$
$$\rightarrow \alpha + \beta < 0 \quad (8)$$



Slika 3.4 Geometrijski uvjet za pripadnost točke P opisanoj kružnici [29]

Budući da su α i β kutovi istog četverokuta njihov zbroj manji je od 2π, uvjet u izrazu (8) možemo preoblikovati kao u izrazu (9).

$$\sin(\alpha + \beta) < 0 \tag{9}$$

Koristeći trigonometrijsku jednakost za sinus zbroja kutova dolazimo do izraza (10).

$$\begin{aligned} &\frac{x_{13}x_{23} + y_{13}y_{23}}{\sqrt{(x_{13}^2 + y_{13}^2)(x_{23}^2 + y_{23}^2)}} \cdot \frac{x_{2P}y_{1P} + x_{1P}y_{2P}}{\sqrt{(x_{2P}^2 + y_{2P}^2)(x_{1P}^2 + y_{1P}^2)}} \\ &+ \frac{x_{13}y_{23} + x_{23}y_{13}}{\sqrt{(x_{13}^2 + y_{13}^2)(x_{23}^2 + y_{23}^2)}} \cdot \frac{x_{2P}x_{1P} + y_{2P}y_{1P}}{\sqrt{(x_{2P}^2 + y_{2P}^2)(x_{1P}^2 + y_{1P}^2)}} < 0 \end{aligned} \tag{10}$$

Gdje su

$x_{13} = x_1 - x_3,$ $x_{23} = x_2 - x_3,$ $x_{1P} = x_1 - x_P,$ $x_{2P} = x_2 - x_P,$

$y_{13} = y_1 - y_3,$ $y_{23} = y_2 - y_3,$ $y_{1P} = y_1 - y_P,$ $y_{2P} = y_2 - y_P$

Iz izraza (10) zaključujemo da se točka P nalazi unutar opisane kružnice ako vrijedi izraz (11).

$$(x_{13}x_{23} + y_{13}y_{23})(x_{2P}y_{1P} - x_{1P}y_{2P}) < (y_{13}x_{23} - x_{13}y_{23})(x_{2P}x_{1P} + y_{1P}y_{2P}) \tag{11}$$

Ako je zadovoljen uvjet iz izraza (11) potrebno je napraviti zamjenu dijagonale. Novonastalim trokutima potrebno je zamijeniti postojeće trokute koji su tvorili četverokut. Također u matrici moramo ažurirati susjedstva susjednim trokutima.

Na kraju triangulacije potrebno je odbaciti trokute koji sadržavaju vrh trokuta koji smo dodali na početku triangulacije, budući da te točke nisu izvorno dio triangulacije.

3.2.1. Ubrzanje algoritma

Kako bi se ubrzala pretraga trokuta pri dodavanju točaka u triangulaciju, autor *S.W. Sloan*[2] predlaže sljedeće korake.

Koordinate točaka normaliziramo na način da im oduzmemo najmanju x odnosno y vrijednost svih točaka triangulacije. Točke zatim dijelimo normalizacijskim faktorom koji računamo prema formuli (12).

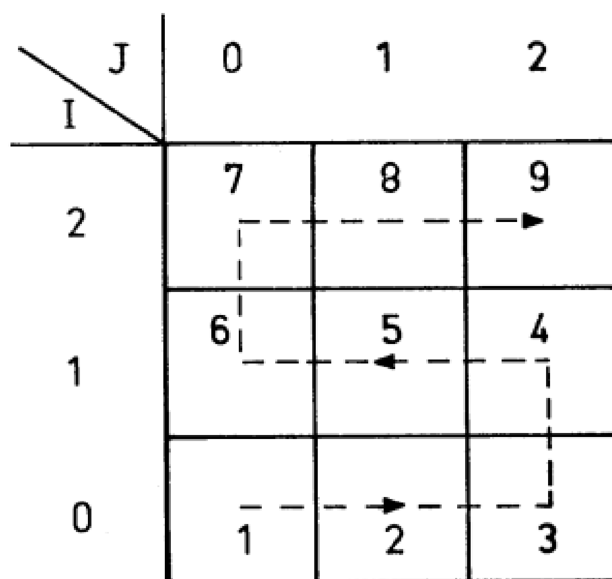
$$s = \max(x_{max} - x_{min}, y_{max} - y_{min}) \tag{12}$$

Gdje su x_{max}, y_{max} najveće, a x_{min}, y_{min} najmanje vrijednosti među koordinatama točaka.

Točkama zatim pridjeljujemo ćelije kao na Slika 3.5 Sortiranje točaka u ćelije. Pripadajuću ćeliju za pojedinu točku određujemo prema izrazu (13).

$$i = 0.99 \cdot n \cdot y$$
$$j = 0.99 \cdot n \cdot x$$
$$BIN = \begin{cases} i \cdot n + j + 1, i\ je\ paran \\ (i+1) \cdot n - j, i\ je\ neparan \end{cases} \tag{13}$$

Gdje je n broj točaka u triangulaciji.



Slika 3.5 Sortiranje točaka u ćelije

Točke zatim sortiramo prema broju pripadajuće ćelije te započinjemo s postupkom dodavanja točaka u triangulaciju.

3.3. Ograničena triangulacija

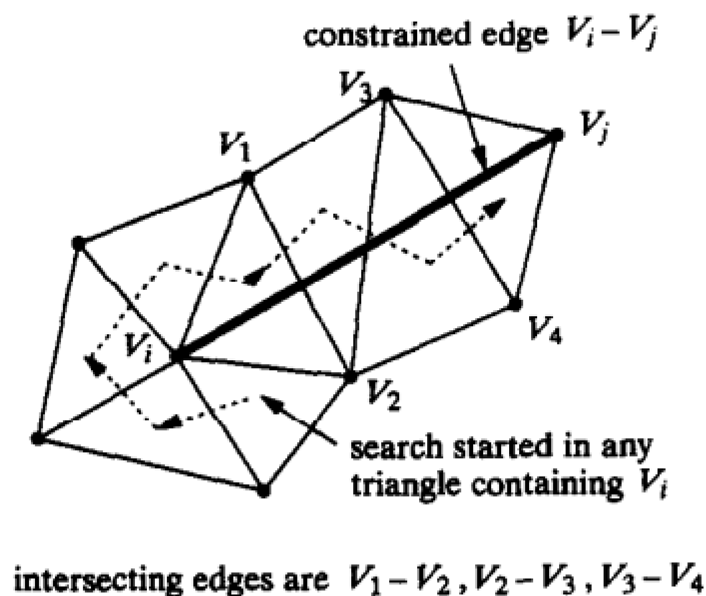
Budući da triangulaciju vršimo nad licem novonastalog fragmenta, potrebno je osigurati da triangulacija uvažava rubove tog fragmenta. Definiramo skup rubova koji čine obrub lica fragmenta nad kojim vršimo triangulaciju te zahtijevamo da ti rubovi budu prisutni u konačnoj triangulaciji. U tu svrhu potrebno je relaksirati uvjet Dealunay triangulacije odnosno izmijeniti trokute triangulacije kako bi se osigurala prisutnost definiranih rubova. Navedeni postupak naziva se ograničena Dealunay triangulacija. Pri implementaciji korišten je algoritam temeljen na radu *A fast algorithm for generating constrained Delaunay triangulations*[3], opis algoritma slijedi u nastavku.

Kao prvi korak algoritma potrebno je napraviti triangulaciju opisanu u prethodnom poglavlju. U idućem koraku za svaki rub ograničenja potrebno je pronaći presjek sa svim stranicama trokuta iz triangulacije. Pretragu započinjemo u početnom vrhu ograničenja te provjeravamo trokute koji sadržavaju taj vrh dok ne pronađemo presjek. Za svaki trokut u pretrazi provjeravamo postoji li presjek jedne od njegovih stranica s ograničenjem. Provjera presjeka stranice trokuta i ograničenja svodi se na provjeru postoji li presjek između dvije dužine. Pri provjeri možemo iskoristiti formulu (7) provjeravajući nalaze li se točke jedne dužine na suprotnim stranama druge dužine. Konačni izraz opisan je formulom (14)

$$\begin{aligned}&\Big(\big(P_{1\times Q} \geq 0 \; and \; P_{2\times Q} \leq 0\big) \; or \; \big(P_{1\times Q} \leq 0 \; and \; P_{2\times Q} \geq 0\big)\Big) \; and \\ &\big((Q_{1\times P} \geq 0 \; and \; Q_{2\times P} \leq 0) \; or \; (Q_{1\times P} \leq 0 \; and \; Q_{2\times P} \geq 0)\big)\end{aligned} \tag{14}$$

Gdje su P_{1xQ} te P_{2xQ} rezultati formule (7) za točke dužine P_1P_2 u odnosu na dužinu Q_1Q_2, a Q_{1xP} te Q_{2xP} rezultati za točke dužine Q_1Q_2 u odnosu na dužinu P_1P_2.

Ako tijekom pretrage otkrijemo trokut čija je stranica jednaka ograničenju, to ograničenje možemo preskočiti budući da ga početna triangulacija već zadovoljava. Nakon pronalaska prvog presjeka pretragu nastavljamo u smjeru drugog vrha ograničenja. Kao idući trokut pretrage odabiremo onaj trokut koji dijeli stranicu koje ograničenje presijeca. Vizualizacija ovog koraka vidljiva je na slici Slika 3.6 Pronalazak presjeka između stranica trokuta i ruba ograničenja.

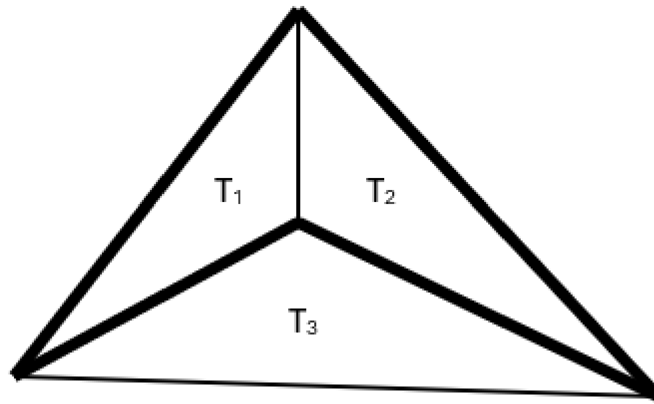


Slika 3.6 Pronalazak presjeka između stranica trokuta i ruba ograničenja[3]

Nakon identificiranja svih stranica koje imaju presjek s ograničenjem, potrebno ih je ukloniti, odnosno izmijeniti trokute kako bi poštovali rub ograničenja. Sve stranice koje presijecaju ograničenje stavljamo na FIFO stog. Za svaku stranicu provjeravamo tvore li dva trokuta koji je dijele konveksan četverokut. Za provjeru je li četverokut konveksan možemo iskoristiti formulu (14), provjeravajući presjek između dijagonala četverokuta. Ako je četverokut konveksan potrebno mu je zamijeniti dijagonalu na način opisan u prethodnom poglavlju, a u protivnom stranicu dodajemo nazad na stog. Za novonastalu dijagonalu također je potrebno provjeriti presijeca li ograničenje. Ako presijeca, dodajemo ju na stog, a u protivnom dodajemo je u listu novih rubova. Ovaj korak ponavljamo sve dok ne otklonimo sve stranice iz stoga.

U idućem koraku iteriramo kroz sve novonastale rubove. Ako je novonastali rub istovjetan ograničenju preskačemo ga. U protivnom provjeravamo zadovoljavaju li trokuti koji ga dijele svojstvo Dealunay triangulacije. Provjeru vršimo koristeći formulu (11) te po potrebi zamjenjujemo dijagonalu na način opisan u prethodnom poglavlju. Ovaj korak ponavljamo sve dok ne dođe do daljnjih zamjena dijagonala.

U posljednjem koraku potrebno je ukloniti sve trokute koji sadržavaju vrh početnog trokuta iz triangulacije ili koji leže izvan granice definirane ograničenjima. Ovaj korak moguće je napraviti iteracijom kroz sve trokute gdje za svaki trokut provjeravamo sadržava li rub ograničenja. Ako sadržava, označavamo kako ćemo zadržati taj trokut te zatim na FIFO stog stavljamo njemu susjedne trokute s kojima ne dijeli rub ograničenja. Svaki trokut na stogu označavamo kako ćemo ga zadržati te na stog dodajemo sve njegove susjede s kojima ne dijeli rub ograničenja. Na ovaj način pronalazimo i obilježavamo kako ćemo zadržati sve trokute unutar prostora ograničenja, dok ostale trokute odbacujemo. Kako bi ovaj postupak bio uspješan, rubovi ograničenja moraju biti definirani u smjeru suprotnom od kazaljke na satu gledajući na ravninu reza. Vizualizacija ovog postupka vidljiva je na slici Slika 3.7 Odbacivanje trokuta T_3 koji se nalazi izvan obruba ograničenja.



Slika 3.7 Odbacivanje trokuta T_3 koji se nalazi izvan obruba ograničenja

3.4. Fragmentacija

Za zadani objekt te željeni broj fragmenata, objekt možemo fragmentirati na način da uzastopno vršimo rezanje objekta opisano u poglavlju 3.1. Kao točku ravnine reza možemo uzeti srednju točku objekta ili točku sudara ako želimo postići dojam lokaliziranog oštećenja. Osim same točke sudara možemo odabrati nasumičnu točku u određenom radijusu od sudara, odnosno točki sudara dodati nasumičan pomak. Razlika između ovih odabira vidljiva je na slici Slika 3.8 Različiti odabiri točke ravnine reza. Za normalu reza možemo generirati nasumični vektor ili ograničiti fragmentiranje na određene osi postavljanjem pripadne komponente vektora normale na nula. Ova mogućnost prikladna je primjerice pri fragmentiranju tankih površina gdje fragmentaciju želimo ograničiti na ravninu površine.

Nakon jednog koraka rezanja nastaju dva nova fragmenta koje stavljamo na FIFO stog. U idućem koraku uzimamo jedan od fragmenata sa stoga te na njemu obavljamo rezanje. Postupak ponavljamo dok ne postignemo željeni broj fragmenata.

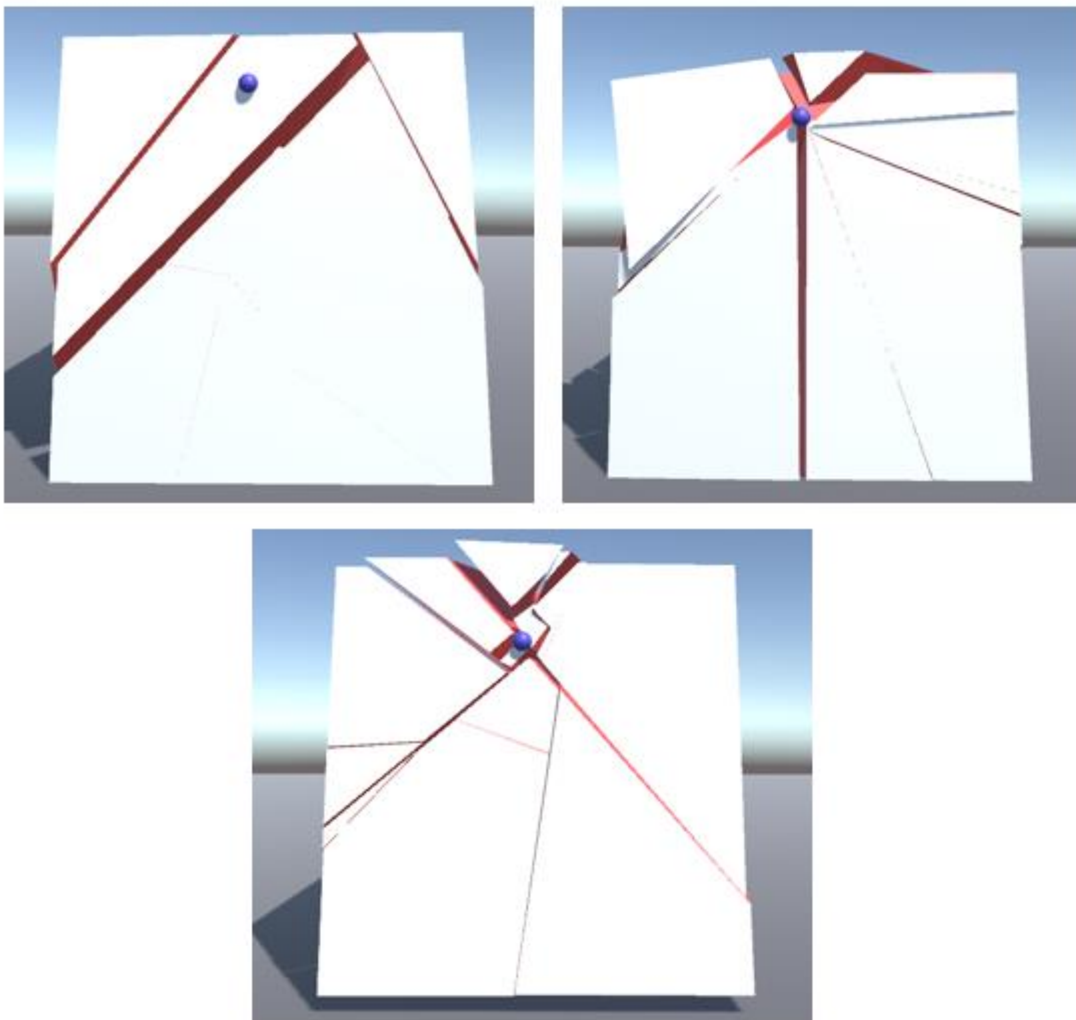
Kako bi objektu dodijelili funkcionalnost fragmentiranja u pogonskom sustavu *Unity*, potrebno mu je dodijeliti pripadnu komponentu. Komponenta odnosno *Component* osnovna je jedinica objekta koja definira njegovo ponašanje te funkcionalnost. Svaki objekt u pogonskom sustavu *Unity* sadržava *Transform* komponentu koja definira poziciju, rotaciju te skalu objekta.

Neke od ostalih osnovnih komponenata uključuju:

- *Mesh Renderer*: omogućava prikaz modela objekta
- *Rigidbody*: omogućava simulaciju fizike nad objektom

- *Collider*: definira granice objekta za sudare

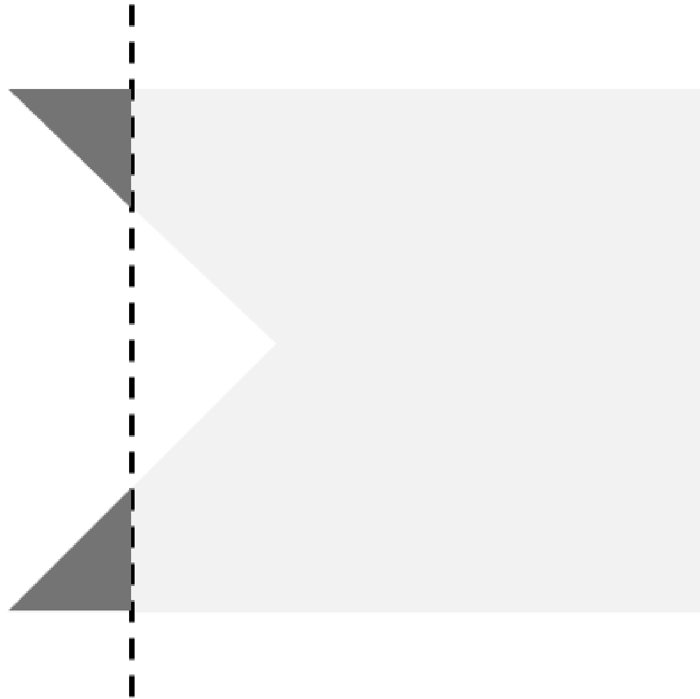
Objektu također možemo dodati proizvoljnu funkcionalnost dodavanjem *Script* komponente. Pri implementaciji definirana je *Fracture Script* komponenta koja omogućava fragmentiranje objekta. Želimo li postići mogućnost ponovnog fragmentiranja nastalih objekata, potrebno im je također dodijeliti *Fracture Script* komponentu. Kako bi ograničili ponovno fragmentiranje, nastalim objektima dodajemo *Fracture Script* komponentu ako zadovoljavaju određen proizvoljan uvjet. Primjeri nekih uvjeta uključuju ne prelazak definiranog broja maksimalne dubine fragmentiranja te minimalna potrebna veličina fragmenta.



Slika 3.8 Različiti odabiri točke ravnine reza

3.5. Razdvajanje otoka

Pri fragmentiranju konkavnog oblika, rezanje oblika može stvoriti fragment koji se sastoji od nekoliko odvojenih dijelova. Te dijelove nazivamo otoci te ih je za ispravnu simulaciju potrebno identificirati i za svaki stvoriti zaseban fragment. Vizualizacija mogućeg nastanka otoka nalazi se na slici Slika 3.9 Nastanak otoka pri rezanju konkavnog oblika.

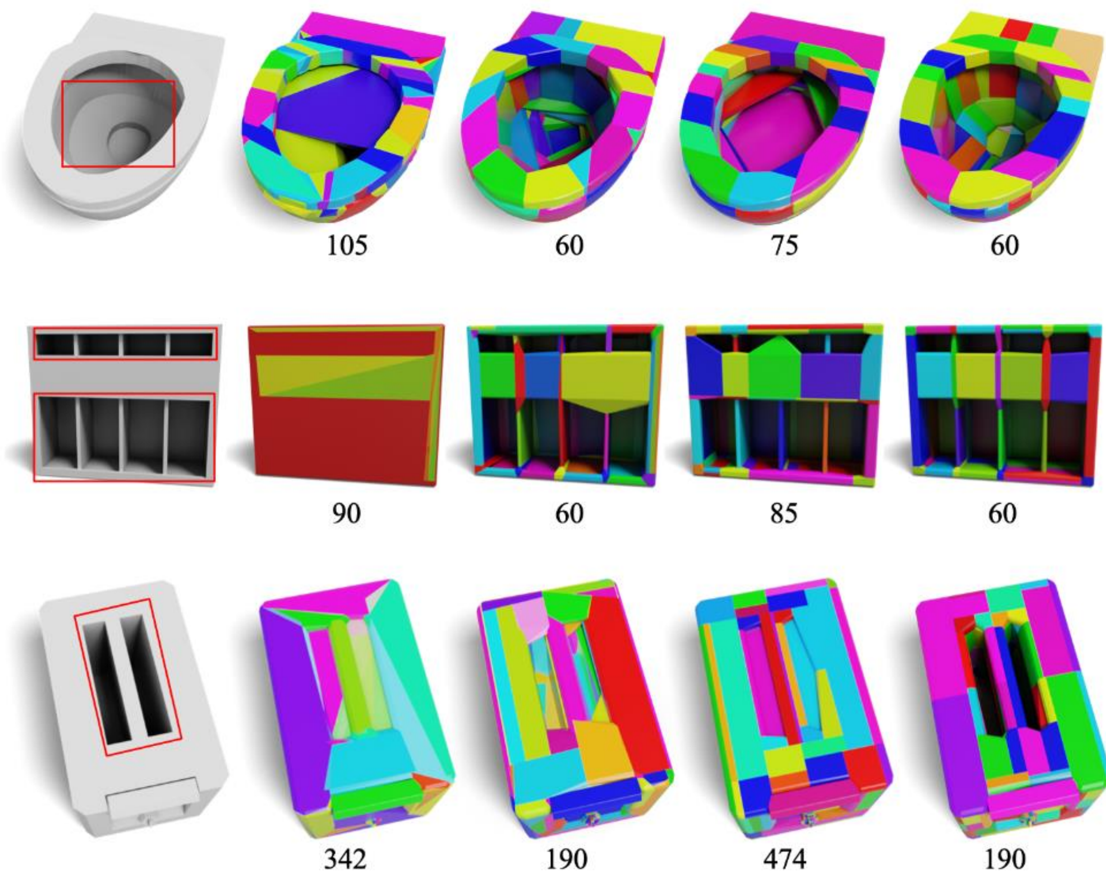


Slika 3.9 Nastanak otoka pri rezanju konkavnog oblika

Pronalazak otoka moguće je napraviti na slijedeći način. Za svaki vrh fragmenta nalazimo sve vrhove podudarne vrhove. Podudarni vrhovi imaju jednake pozicije te se nalaze na granicama između lica geometrije fragmenta kako bi se za svako lice mogle ispravno odrediti vrijednosti normala i UV koordinata. Nakon pronalaska podudarnih vrhova, svakom vrhu pridjeljujemo sve trokute kojima pripada. U idućem koraku iteriramo kroz sve vrhove. Za svaki vrh ako već nije posjećen dodajemo ga na stog pretrage. Iteriramo dok ne ispraznimo stog pretrage. Skidamo vrh sa stoga pretrage te ga označavamo posjećenim. Na stog pretrage zatim dodajemo vrhove svih trokuta kojima vrh sa stoga pripada, te njima podudarne vrhove. Kada ispraznimo stog pretrage pronašli smo sve vrhove koji pripadaju otoku. Za pronađene vrhove definiramo novu mrežu poligona te novi fragment.

3.6. Konveksna dekompozicija

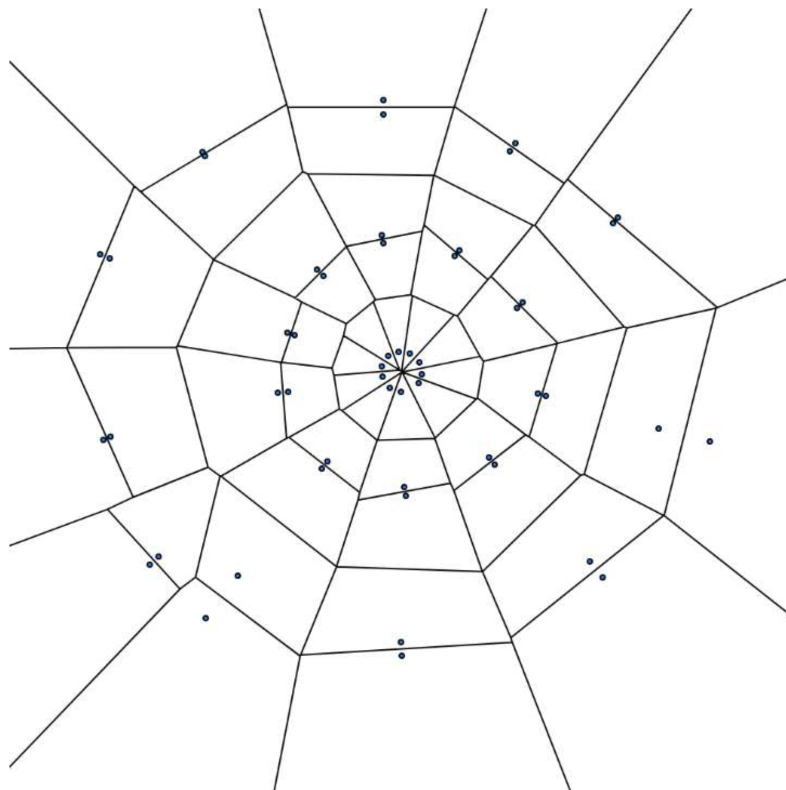
Postupak pronalaska te razdvajanja otoka računalno je zahtjevan te bi ga stoga bilo bolje izbjeći. Budući da rezanjem konveksnih oblika ne nastaju otoci, pronalazak otoka ne bi trebali raditi kada bi konkavne oblike razdvojili na više manjih konveksnih oblika. Navedeni postupak naziva se konveksna dekompozicija. Konveksna dekompozicija je NP-težak problem no moguće ju je napraviti prije same simulacije. Na ovaj način smanjuju se računalni resursi pri simulaciji ali se dodaje dodatan korak pri pripremi objekata čime se umanjuje prednost fragmentacije u stvarnom vremenu u odnosu na pristup zamjene fragmentiranim modelom. Različite rezultate konveksne dekompozicije ovisno o implementaciji može se vidjeti na slici Slika 3.10 Rezultati različitih pristupa implementacije konveksne dekompozicije



Slika 3.10 Rezultati različitih pristupa implementacije konveksne dekompozicije [5]

4. Uzorak prijeloma

Određeni materijali poput stakla imaju karakterističan uzorak prijeloma pri sudaru s objektom. Za modeliranje takvog prijeloma možemo koristiti Voronoi dijagram. Voronoi dijagram za skup točaka razdvaja ravninu na konveksne poligone, pri čemu svaki poligon predstavlja područje, zvano Voronoi ćelija, u kojem su sve točke bliže određenoj točki iz skupa nego bilo kojoj drugoj točki iz tog skupa. Budući da uzorak prijeloma ovisi samo o materijalu moguće ga je unaprijed izračunati te primijeniti na prikladne objekte pri simulaciji. Slika prikazuje modelirani Voronoi dijagram za fragmentiranje stakla.



Slika 4.1 Voronoi dijagram za fragmentiranje stakla

Kako bismo primijenili uzorak prijeloma potrebno ga je prikladno skalirati te po potrebi translatirati kako bi se poravnao s izvorom oštećenja. U ovom koraku moguće je napraviti i druge translacije poput rotacije. U idućem koraku površinu objekta razdvajamo na fragmente temeljem ćelija Voronoi dijagrama. Za svaku ćeliju dijagrama potrebno je odrediti pripadne točke odnosno rubove površine objekta. Za svaki rub površine objekta provjeravamo nalaze li se unutar Voronoi ćelije.

Ako su rubovi Voronoi ćelije opisani u smjeru suprotnom od kazaljke na satu, provjera nalazi li se rub površine unutar ćelije svodi se na provjeru nalazi li se s lijeve strane svih rubova ćelije.

Za Voronoi dijagram generiran za više od dvije točke vrijedi da su rubovi svake njegove ćelije dužine ili polupravci. Polupravac možemo opisati izrazom (15)

$$\vec{R} = \overrightarrow{P_0} + t\vec{d} \tag{15}$$

gdje je P_0 početna točka polupravca, d vektor smjera, a t parametar koji je veći od nula.

Provjeru nalazi li se vrh ruba površine s lijeve strane ruba ćelije možemo napraviti koristeći formulu (7) za rubove ćelije koji su dužine. Dok istu provjeru za rubove ćelije koji su polupravci možemo napraviti koristeći formulu (16)

$$d = (P_{1x} - P_{0x})d_y - (P_{1y} - P_{0y})d_x \tag{16}$$

Gdje je P_1 točka za koju radimo provjeru, a P_0 i d točka i vektor smjera polupravca. Točka se nalazi s lijeve strane polupravca ako je vrijednost izraza veća od nula.

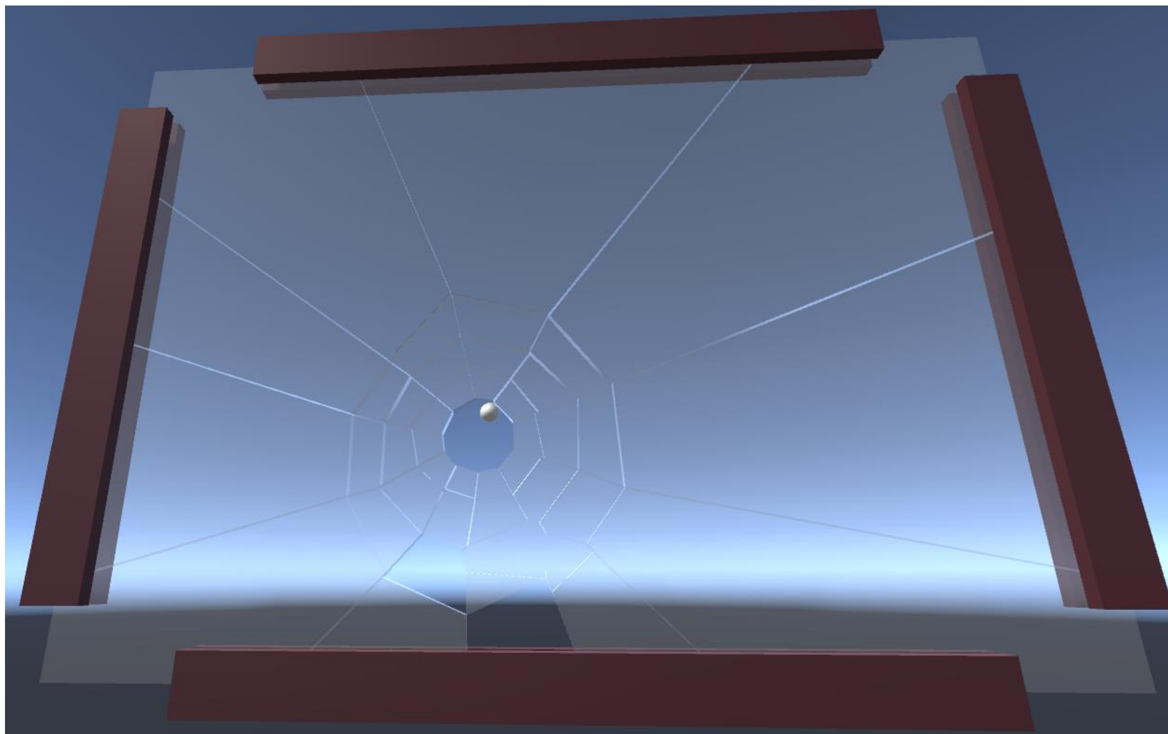
Provjeru radimo za obje točke ruba površine te razlikujemo slučajeve:

1. Obje točke nalaze se s lijeve strane
2. Obje točke nalaze se s desne strane
3. Jedna od točki nalazi se s lijeve a druga s desne strane

Pri slučaju 1 rub se nalazi s lijeve strane ruba ćelije te ga je potrebno dodatno provjeriti s ostalim rubovima ćelije. Pri slučaju 2 rub se nalazi s desne strane ruba ćelije te ga stoga odbacujemo budući da se ne nalazi u ćeliji. Pri slučaju tri postoji presjek ruba s rubom ćelije te je potrebno pronaći točku presjeka. Točku presjeka možemo pronaći koristeći formulu (). U ovom slučaju definiramo novi rub koji se sastoji od točke s lijeve strane ruba ćelije te točke presjeka s rubom ćelije. Nastali rub je potrebno provjeriti s ostalim rubovima ćelije.

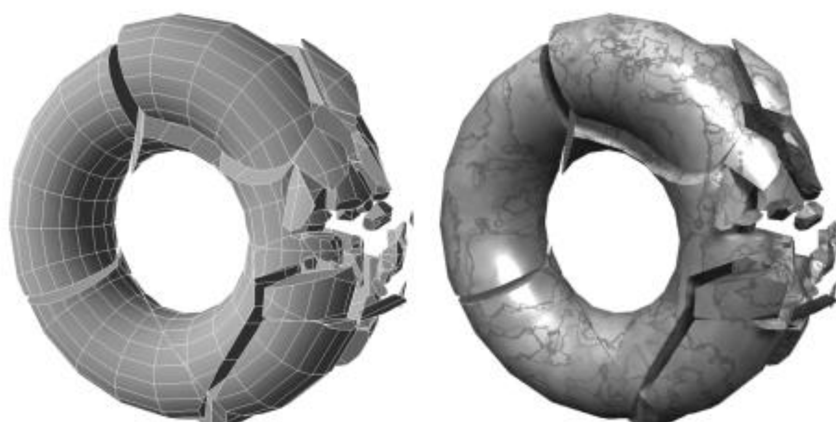
Nakon završetka ovog koraka nastaje poligon u obliku Voronoi ćelije. Iz nastalog poligona potrebno je definirati točke geometrije za novi fragment. Ako je n broj vrhova poligona, potrebno je definirati 6n vrhova geometrije za novi fragment. Pri implementaciji korištena je pretpostavka da je oblik kojeg fragmentiramo konstante dubine d. 3D vrhove geometrije tada možemo definirati koristeći 2D koordinate vrha poligone te postavljajući z koordinatu na vrijednost d/2 odnosno -d/2.

Rezultat primjene uzorka prijeloma stakla vidljiv je na slici Slika 4.2 Primjena uzorka prijeloma stakla.



Slika 4.2 Primjena uzorka prijeloma stakla

Ovaj postupak netrivijalno je proširiv na 3D Voronoi dijagrame, čijim je korištenjem moguće pobliže definirati 3D oblike nastalih fragmenata. Rezultate takvog pristupa moguće je vidjeti na slici Slika 4.3 Fragmenti nastali primjenom 3D Voronoi dijagrama



Slika 4.3 Fragmenti nastali primjenom 3D Voronoi dijagrama [4]

5. Simulacija fizike fragmenata

Kako bi nad fragmentima mogli simulirati fiziku u programskom pogonu *Unity*, potrebno im je definirati *Rigidbody* komponentu. Dodatno potrebno je dodati *Collider* komponentu kako bi se za objekt detektirali sudari.

Rigidbody komponenta sadrži svojstva poput brzine, mase, otpora koja se koriste pri izračunu sila na objektu. Većinu ovih svojstava možemo kopirati iz originalnog fragmenta dok masu novonastalih fragmenata možemo izračunati na temelju izračunate gustoće originalnog fragmenta. Gustoću možemo izračunati formulom (17)

$$d = \frac{B_{0x} \cdot B_{0y} \cdot B_{0z}}{m_0} \tag{17}$$

Gdje su B_{0x}, B_{0y}, B_{0z} komponente obujmice originalnog fragmenta, a m_0 masa originalnog fragmenta. Masu novonastalog fragmenta možemo izračunati formulom (18)

$$m = \frac{B_x \cdot B_y \cdot B_{0z}}{d} \tag{18}$$

Gdje su B_x, B_y, B_z komponente obujmice fragmenta, a d izračunata gustoća.

6. Rezultati

Sva mjerenja izvršena su na konfiguraciji računala:

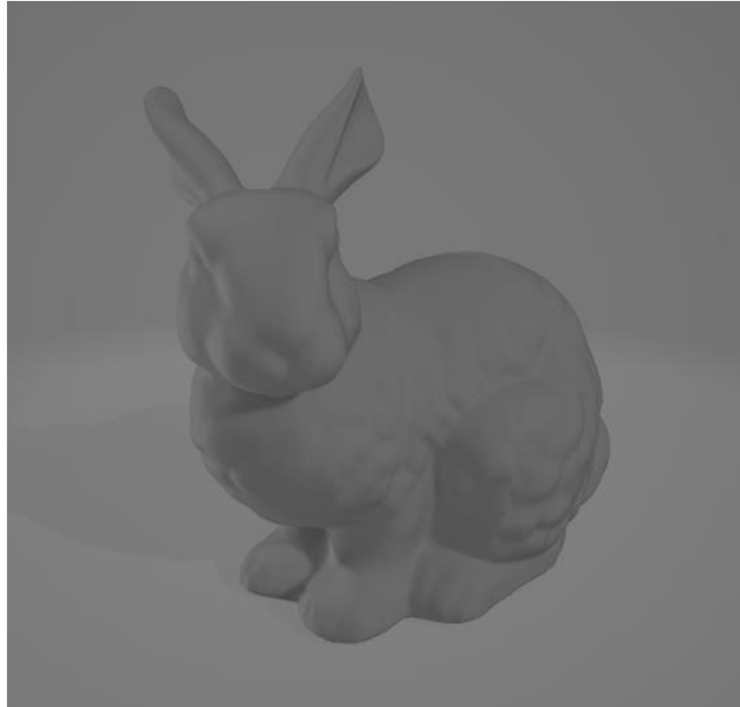
- Procesor: AMD Ryzen 5 3600 6-Core Processor, 3.6 GHz
- Grafička kartica: Radeon RX 580, 4GB (driver verzija: 31.0.21916.2)
- RAM: 16GB DDR4, 3200MHz
- Operacijski sustav: Windows 10

Kako bi odredili performanse programa mjerimo vrijeme potrebno za izračun slike. Ova mjera obrnuto je proporcionalna mjeri slika po sekundi (*eng. frames per second* - *FPS*) koja se često koristi pri utvrđivanju performansi grafičkih programa. FPS ne mjerimo iz razloga što je pri izvođenju simulacije bio konstantne zadane maksimalne vrijednosti 300, osim u trenutku izračuna fragmentacije. Kao mjera performanse simulacije stoga je odabrano vrijeme izračuna slike u trenutku fragmentacije.

Pri mjerenju korišten je model *Stanford Bunny* različite kompleksnosti, odnosno različitih brojeva vrhova te trokuta. Izgled modela može se vidjeti na slici Slika 6.1 Korišteni model pri mjerenju. Podaci o korištenim modelima pri mjerenju prikazani su u tablici Tablica 6.1 Podaci o korištenim modelima.

Naziv modela	Broj vrhova	Broj trokuta
Reduced	22688	8754
Small	20569	35032
Medium	40552	70064
Large	158505	280256

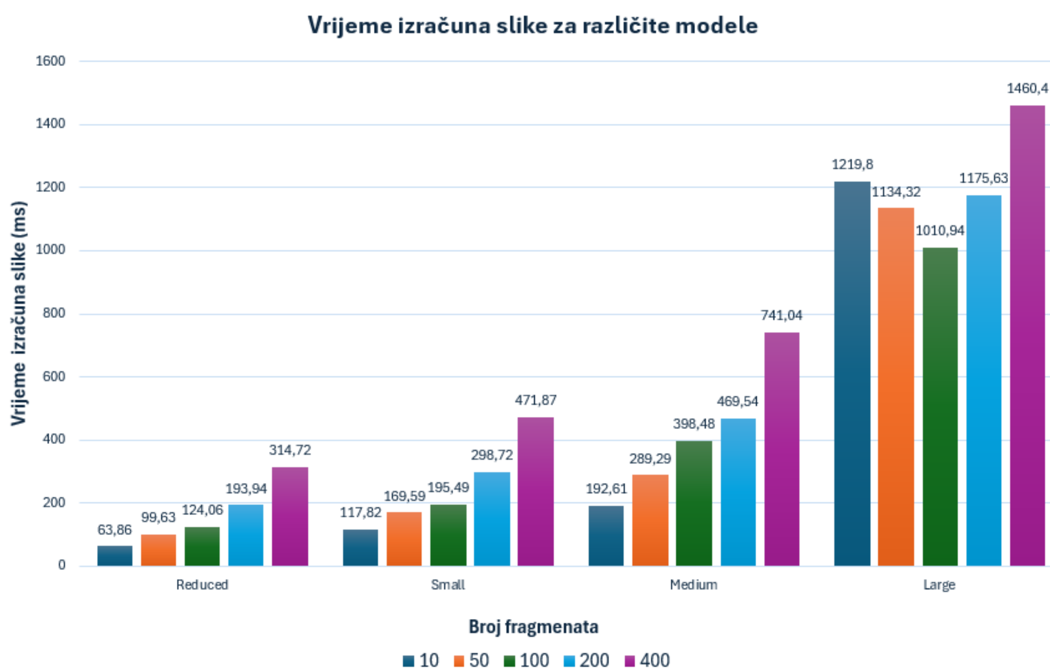
Tablica 6.1 Podaci o korištenim modelima



Slika 6.1 Korišteni model pri mjerenju

Vremena izračuna slike za različite modele te različit broj fragmenata prikazana su na slici Slika 6.2 Vrijeme izračuna slike za različite modele. Izmjereni podaci prikazuju kako je vrijeme izračuna proporcionalno broju fragmenata koji nastaju u simulaciji, što je očekivano. Jedino odstupanje vidljivo je za najveći model, *Large*, gdje je najmanje vrijeme zabilježeno pri simulaciji s 100 fragmenata. Razlog ovome je vrijeme potrebno za stvaranje *Unity* objekata za nastale fragmente. Budući da je za simulaciju fizike potrebna *Collider* komponenta potrebno ju je izračunati za svaki novi objekt. *Unity* automatski izračunava vrhove *Collider* komponente koristeći vrhove mreže poligona objekta. Zbog kompleksnosti *Large* modela te malog broja fragmenata nastali fragmenti sadržavati će značajan broj vrhova te trokuta što usporava stvaranje *Collider* komponente. Vrijeme izračuna također je proporcionalno kompleksnosti korištenog modela što je vidljivo iz rezultata mjerenja.

Ukupan broj nastalih vrhova te trokuta nakon fragmentacije prikazani su u tablicama Tablica 6.2 Ukupan broj vrhova nastalih fragmenata za različite modele te broj fragmenata te Tablica 6.3 Ukupan broj trokuta nastalih fragmenata za različite modele te broj fragmenata.



Slika 6.2 Vrijeme izračuna slike za različite modele

	1	10	50	100	200	400
Reduced	22688	31723	54926	67694	108478	163241
Small	20569	37607	69218	98404	158111	236891
Medium	40552	65535	105768	150476	223624	346164
Large	158505	196506	265881	347452	458298	646530

Tablica 6.2 Ukupan broj vrhova nastalih fragmenata za različite modele te broj fragmenata

	1	10	50	100	200	400
Reduced	8754	13870	26962	34036	56886	87458
Small	35032	44724	62610	79058	112738	156962
Medium	70064	84284	107118	132434	473784	242874
Large	280256	301928	341524	387736	458298	646530

Tablica 6.3 Ukupan broj trokuta nastalih fragmenata za različite modele te broj fragmenata

Zaključak

Simulacija lomljivih objekata opisuje raznolik problem simuliranja destrukcije okoliša i objekata virtualnog svijeta. Zbog zahtjeva za izvođenjem u stvarnom vremenu realističnost simulacije potrebno je podrediti performansama izvođenja programa.

U ovom radu prikazane su različiti pristupi simulacije lomljivih objekata te njihove prednosti, nedostaci te područja primjene. Za simulaciju površinskih oštećenja prikladno je primijeniti teksture oštećenja dok za simulaciju uništenja samog objekta odabir konkretnog pristupa uvelike ovisi o potrebama te kontekstu samog programa u kojem se implementira.

Uzorak prijeloma dozvoljava simulaciju karakterističnih prijeloma materijala poput stakla, a zbog mogućnosti izračuna Voronoi dijagrama prije same simulacije, nije uvelike računski zahtjevan.

Zamjena oštećenim modelom omogućava vrlo laku implementaciju te je najmanje računski zahtjevna, ali ne nudi mogućnost prikaza lokalnog oštećenja.

Dinamička fragmentacija nudi najfleksibilniji pristup omogućavajući prilagodbu same fragmentacije na uvjete u virtualnom svijetu. Zbog zahtjeva za računalnim resursima često nije primjenjiva na modele s velikim brojem trokuta ili u programima koji su već računalno zahtjevni.

Zamjena fragmentiranim modelom nudi razumnu ravnotežu između performansi te vizualnih rezultata što je čini dobrim izborom ako nije potrebno simulirati lokalna oštećenja.

Literatura

[1] Dobransk´y, M. *Efficient simulation of environment destruction in games*. Bachelor thesis. Charles University Department of Software Engineering, 2017.

[2] Sloan, S.W. *A fast algorithm for constructing Delaunay triangulations in the plane* The University of Newcastle Department of Civil Engineering and Surveying, 1987.

[3] Sloan, S.W. *A fast algorithm for generating constrained Delaunay triangulations* The University of Newcastle Department of Civil Engineering and Surveying, 1992.

[4] Muller, M., Chentanez N, Kim T.Y., *Real Time Dynamic Fracture with Volumetric Approximate Convex Decompositions*, ACM Transactions on Graphics, 2013.

[5] Xinyue, W., Minghua L., Zhan L., Hao S., *Approximate Convex Decomposition for 3D Meshes with Collision-Aware Concavity and Tree Search*, ACM Transactions on Graphics, 2022.

Sažetak

Simulacija lomljivih objekata u stvarnom vremenu

Interakcija korisnika s okolišem kroz simulaciju lomljivih objekata igra veliku ulogu u ostvarivanju uranjanja korisnika u virtualni svijet. Popularno područje primjene simulacije lomljivih objekata u stvarnom vremenu su računalne igre. Zbog potrebe za simulacijom u stvarnom vremenu često je potrebno relaksirati zahtjeve za realističnost simulacije u korist efikasnijih metoda. Kompleksnost implementacije simulacije također uvelike ovisi o kontekstu te potrebama virtualnog svijeta u kojem će se primjenjivati. Ovaj rad daje pregled različitih metoda simulacije lomljivih objekata u stvarnom vremenu. Za određene pristupe napravljena je te opisana implementacija u pogonskom sustavu *Unity*.

Ključne riječi: lomljivi objekti, simulacija, rezanje mreže poligona, triangulacija

Summary

Real Time Simulation of Destructible Objects

User interaction with the environment through simulation of destructible objects plays a major role in immersing the user in the virtual world. A popular field of application of real time simulation of destructible objects are computer games. Due to a constraint of real time simulation, it is often necessary to relax the requirements for simulation realism in favour of more efficient methods. The complexity of the simulation is also largely dependent on the context and the needs of the virtual world in which it will be applied. This thesis provides an overview of different approaches of real time simulation of destructible objects. For certain methods, a *Unity* implementation is provided and analysed.

Keywords: destructible objects, simulation, mesh cutting, triangulation

Skraćenice

LIFO	*Last in, first out*	zadnji ulazi, prvi izlazi
VR	*Virtual Reality*	virtualna stvarnost
FIFO	*First in, first out*	prvi ulazi, prvi izlazi

Privitak

Implementacija rada dostupna je na poveznici: https://github.com/Vercy09/FractureSim

Prije instalacije potrebno je instalirati Unity 2022.3.20f1.