SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 4797

Simulacija lomljenja objekata

Josip Sito

Zagreb, lipanj 2017.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA ZAVRŠNI RAD MODULA

Zagreb, 3. ožujka 2017.

ZAVRŠNI ZADATAK br. 4797

Pristupnik: **Josip Sito (0036487715)**
Studij: Računarstvo
Modul: Programsko inženjerstvo i informacijski sustavi

Zadatak: **Simulacija lomljenja objekata**

Opis zadatka:

Proučiti načine na koje će se objekt lomiti uslijed udarca. Obratiti pažnju na različite materijale i svojstva objekata. Razraditi simulaciju pucanja i lomljenja objekta uslijed udarca drugog objekta. Na različitim primjerima prezentirati ostvarene rezultate. Diskutirati utjecaj parametara. Načiniti ocjenu rezultata i implementiranih algoritama.
Izraditi odgovarajući programski proizvod. Koristiti grafički programski alat Blender te grafički pogon Unity. 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.

Zadatak uručen pristupniku: 10. ožujka 2017.
Rok za predaju rada: 9. lipnja 2017.

Mentor:
Prof. dr. sc. Željka Mihajlović

Djelovođa:
Doc. dr. sc. Mirjana Domazet-Lošo

Predsjednik odbora za završni rad modula:
Doc. dr. sc. Ivica Botički

SADRŽAJ

1. Uvod

Pošto su u stvarnom svijetu sve stvari lomljive, došlo je do ideje da se i u računalnoj grafici pokuša imitirati takvo fizikalno ponašanje. Sustavi koji pružaju točan prikaz simulacije fizike lomljenja objekata zahtijevaju mnogo računalne snage, neisplativi su i teško ih je implementirati te se zbog toga koriste razne metode koje čine sustav simulacije fizikalno nepreciznim, ali unatoč tome nude prihvatljivo rješenje u kontekstu računalnih igara [1]. U početnom poglavlju opisuju se osnovne značajke različitih načina lomljenja tijela u računalnoj grafici. Zatim ovaj rad opisuje matematičke osnove Delanauy–ove triangulacije koja je bitna za shvaćanje implementacije simulacije lomljenja u grafičkom pogonu Unity.

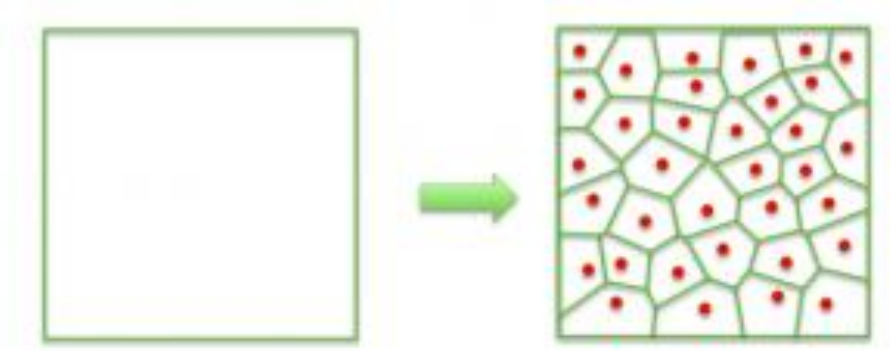
2. Načini lomljenja tijela

U ovom poglavlju obratit ćemo pažnju na osnovne karakteristike poznatijih načina lomljenja tijela, a u idućim poglavljima detaljnije su objašnjeni Voronoi-ev lom i tetraedarizacija. Standardni pristup lomljenju objekata može se podijeliti na fazu pripreme 3D geometrije i na fazu lomljenja objekata u stvarnom vremenu [2].

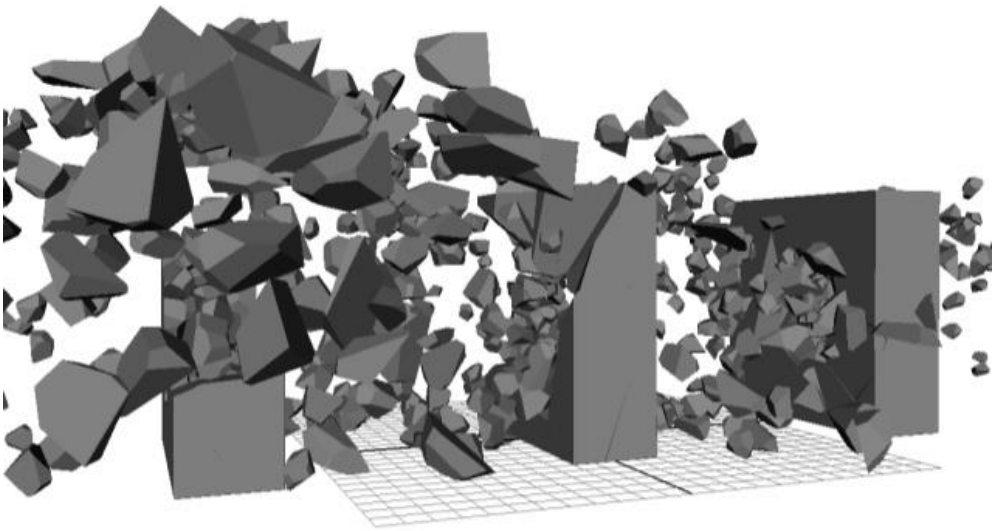
2.1 Priprema 3D geometrije

2.1.1. Voronoi-ev lom

Vokonoi-ev lom se zasniva na principu Voronoi-evog grafa koji dijeli ravninu na regije temeljene na blizini točaka određenom podskupu ravnine (Slika 2.1). Unutar 3D modela se zadaju točke i prema njima se stvaraju Voronoi-eve regije koje ograđuju određenu točku (Slika 2.2). Jedan od poznatijih algoritama koji određuju Voronoi-ev graf na temelju zadanih točaka je Fortune-ov algoritam [2]. Upravo zbog vizualnog izgleda loma i zbog algoritama koji su optimizirani, ovaj način lomljenja objekata je vrlo popularan i koriste ga poznatiji grafički alati kao npr. Unreal Engine, Maya, Blender.



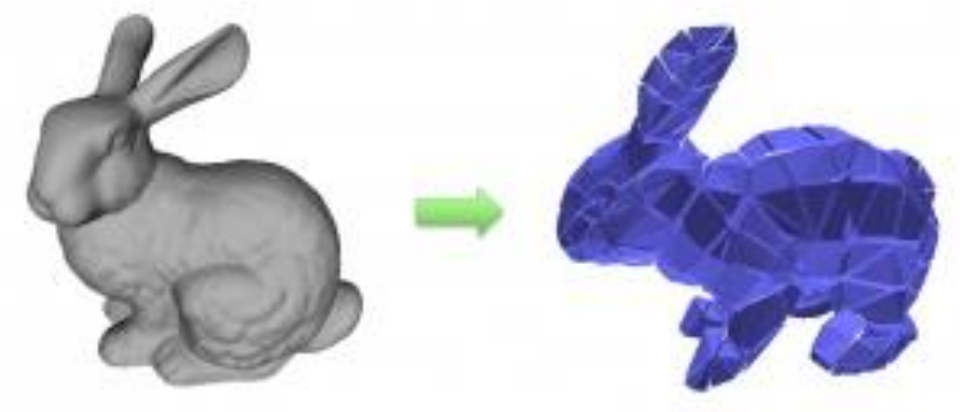
Slika 2.1 Podjela ravnine na Voronoi regije



Slika 2.2 Primjer Voronoi loma u 3D grafičkom alatu

2.1.2. Tetraedarizacija

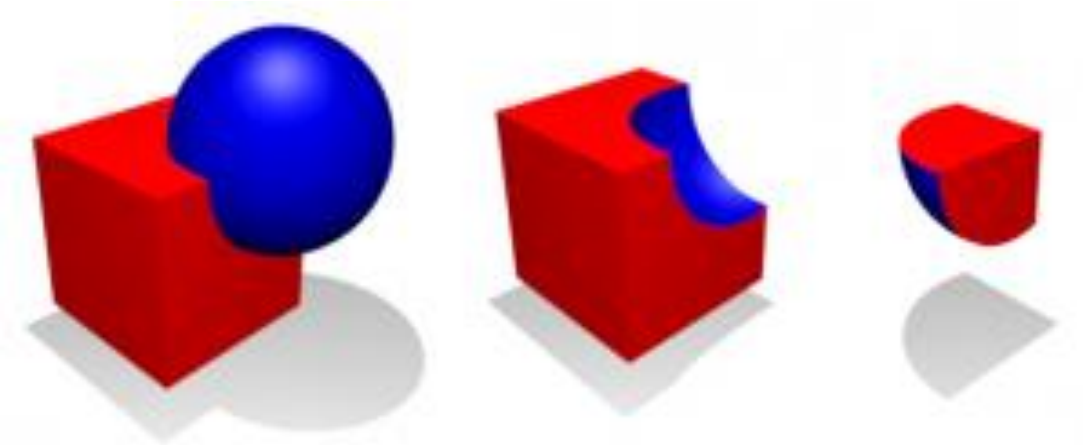
Tetraedarizacija je postupak koji dijeli određeni 3D objekt na skup tetraedara (Slika 2.3). Tetraedarizacija se temelji na Delaunay-ovoj triangulaciji koja je dualna s Voronoi-evim grafom. Pojedine otvorene knjižnice koje implementiraju ovaj način lomljenja su NetGen, TetGen i MIConvexHull.



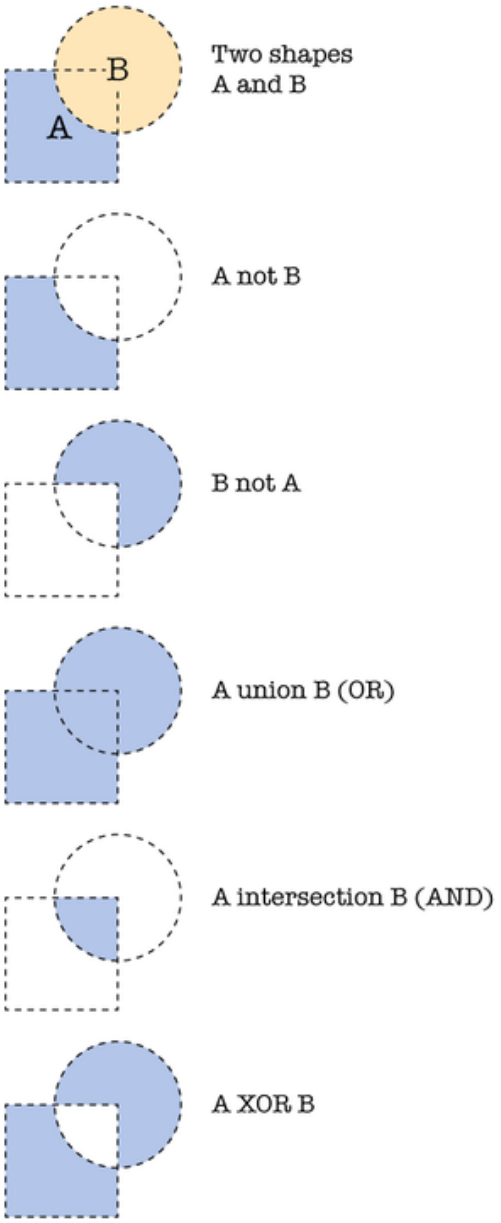
Slika 2.3 Primjer tetraedarizacije na 3D modelu zeca.

2.1.3. Boolean-ove operacije

Boolean-ove operacije ili CSG (*eng. Constructive Solid Geometry*) je način izvođenja volumnih operacija nad jednostavnim 3D objektima kako bi se dobili složeniji oblici objekata ili kako bi se objekti razlomili na manje dijelove (Slika 2.4). Neke od osnovih operacija su unija, razlika i presjek (Slika 2.5).



Slika 2.4 Podjela kocke na 2 dijela uz korištenje Boolovih operacija



Slika 2.5 Prikaz boolean-ovih operacija nad objektima kugle i kocke, Plavom bojom je prikazan rezultat određene operacije

2.2. Lomljenje objekata u stvarnom vremenu

2.2.1. Boolean-ove operacije u stvarnom vremenu

Zbog jednostavne implementacije CSG se mogu koristiti u stvarnom vremenu kako bi se dobila bolja prezicnost. No ukoliko operaciju obavljamo sa složenim objektima koji sadrže velik broj poligona, trebat će nam puno procesorske snage. Kako bi se uštedilo na vremenu potrebnome za računanje boolean-ove operacije, u praksi se koriste oktalno[1] ili BSP[2] stablo [3]. Primjer takvog lomljenja prikazuje se u videoigri „Red faction“ (Slika 2.6).

Slika 2.6 Uništavanje zida u videogri „Red faction“

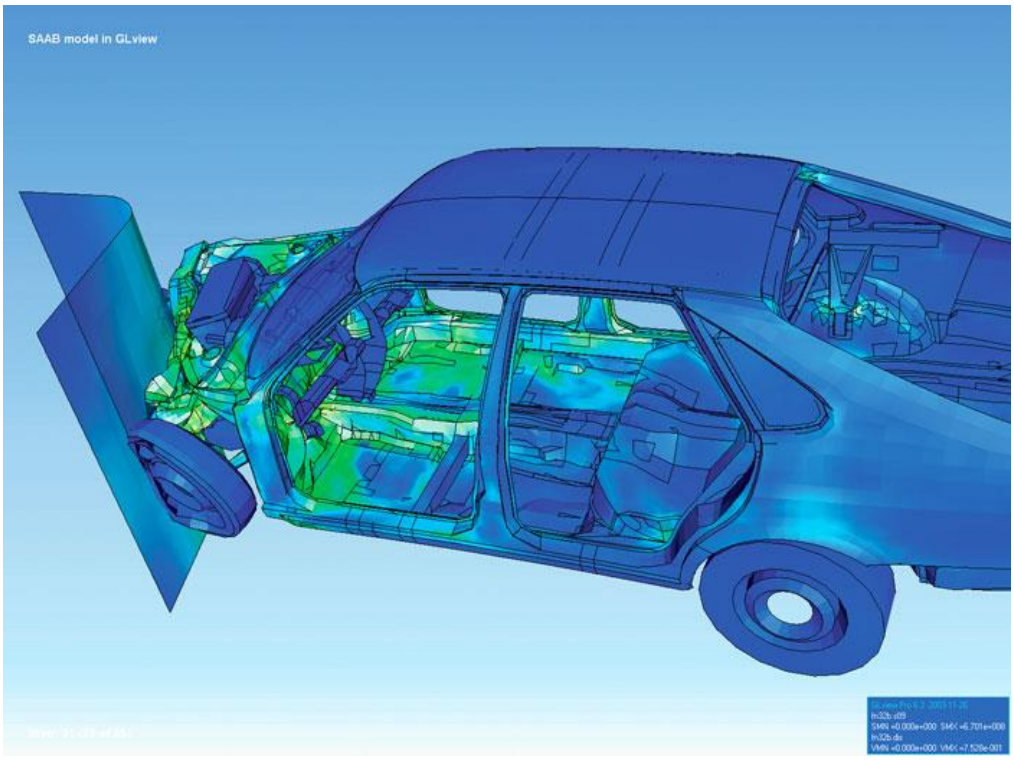
2.2.2. Metoda konačnih elemenata

Metoda konačnih elemenata koristi znanja iz mehanike neprekidnih sredina (mehanike kontinuuma) kako bi simulirala deformacije i lomove objekata (Slika 2.7). Ideja metode konačnih elemenata je da se određeno područje podijeli na manja podpodručja koja se nazivaju konačni elementi. Elementi su međusobno povezani s točkama koje nazivamo čvorovi. Metodom konačnih elemenata, kao i mnogim drugim numeričkim metodama, omogućeno je izbjegavanje integriranja koje je računski vrlo zahtjevno. Ono što metodu konačnih elemenata čini vrlo povoljnom za probleme unutar računalne grafike je to što se jednadžbe konačnih elemenata mogu zapisati u obliku velike matrice koja sadrži globalnu jednadžbu konačnih elemenata. Matrice su

[1] Stablasta struktura podataka čiji čvor sadrži podjelu na osam grana koje dalje rekurzivno pratimo. Koristi se u računalnoj grafici za hijerarhijski prikaz 3D podataka kod uklanjanja skrivenih linija i površina. [7]

[2] (*eng. Binary Space Partitioning*) Organizacija podataka koja se koristi u računalnoj grafici kod uklanjanja skrivenih linija i površina. Osnovna ideja izgradnje BSP stabla je rekurzivna binarna podjela prostora. [7]

iznimno povoljne za implementaciju paralelizma, čijom primjenom rješavamo vječni problem računalne grafike, a to su resursi [3].



Slika 2.7 Simulacija deformacije auta tijekom sudara korištenjem metode konačnoh elemenata

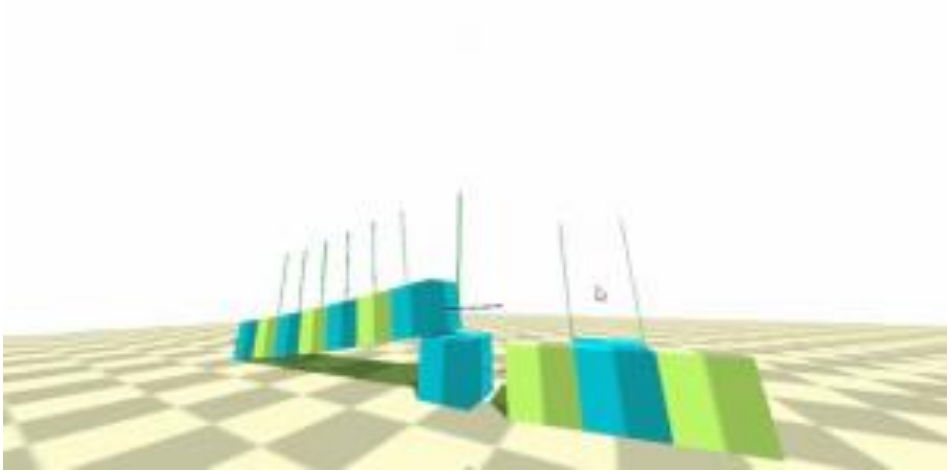
2.2.3. Postavljanje lomljivog kompozitnog čvrstog tijela

Kako bi koristili ovu metodu prvo se određuje 3D geometrija u polomljenom obliku te se dijelovi povezuju u jedno čvrsto tijelo. Veze se mogu ostvariti na više načina. Jedan od načina je definiranje veza između svih odlomljenih dijelova tijela. Time dobivamo najviše kontrole, ali gubimo na performansama zbog velike količine veza. Drugi način je da se veze automatski izračunaju na temelju dodirnih točaka između dijelova koji se dodiruju. Nakon definiranja veza, tim vezama dodijeljujemo određeni prag lomljenja. Ako bi tijekom simulacije došlo do sudara, računamo impuls sile. Ukoliko je impuls sile veći od praga kojega smo prethodno odredili, veza između objekata slabi ili se lomi (briše). Svakom objektu koji je odvojen od početne kompozicije čvrstog tijela dodjeljuje se novo čvrsto tijelo (Slika 2.8) [2].

Slika 2.8 Prikaz lomljenja kompozitnog čvrstog tijela

2.2.4. Postavljanje lomljivih veza između čvrstih tijela

Za razliku od prethodne metode u ovoj se definira čvrsto tijelo za svaki polomljeni objekt. Između čvrstih tijela postavljamo lomljive veze kojima definiramo prag lomljenja. Razlika između ove i prethodne metode je u tome što veze u ovoj metodi neće biti toliko krute te će doći do savijanja kod povezanih objekata (Slika 2.9). [2]



Slika 2.9 Prikaz lomljenja čvrstih tijela povezanih lomljivim vezama

2.2.5. Hibridne metode

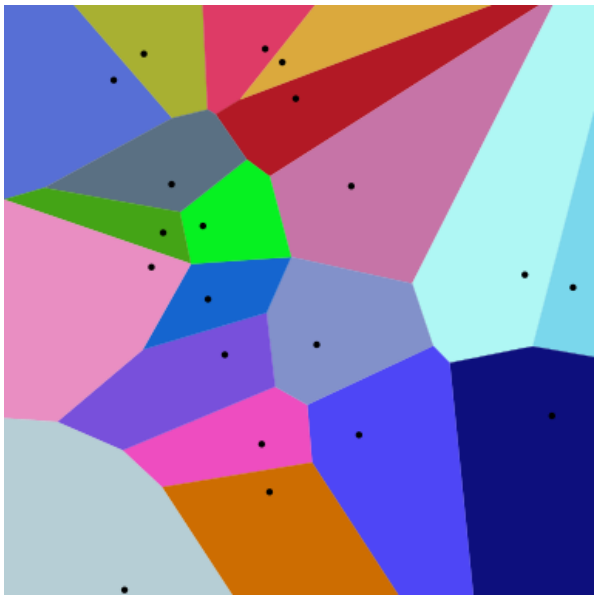
Također je moguće sve te metode kombinirati. Npr. moguće je koristi metodu konačnih elemenata kako bi se odredila svojstva loma, a zatim postaviti čvrsta tijela [4].

3. Matematička pozadina Delanauy-ove triangulacije

Kako se u simulaciji opisanoj u 4. poglavlju koristi implementacija Delaunay-ove triangulacije, sljedećih par poglavlja opisuju matematičke osnove Delaunay-ove triangulacije i njemu dualnog Voronoi-evog grafa.

3.1. Voronoi-ev dijagram

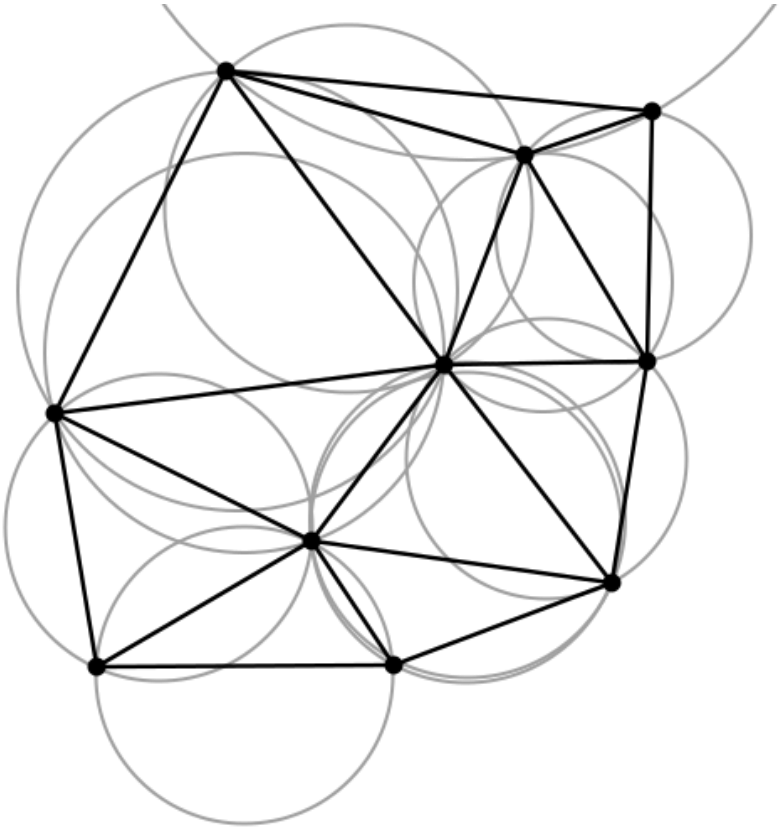
Dan je skup S s n točaka u ravnini R^2 , za svaku točku p_i iz skupa S postoji skup točaka ravnine koje su bliže točki p_i od bilo koje druge točke iz skupa S. Skup takvih točaka zove se Voronoi-ev poligon. Skup n Voronoi-evih poligona za n zadanih točaka čini Voronoi-ev dijagram (Slika 3.1). Za dvije točke možemo odrediti pravac koji predstavlja skup točaka jednako udaljenih od te dvije točke. Sjecište dva pravca je jednako udaljeno od tri promatrane točke (opisana kružnica trokuta). Proširenjem na 3D prostor ćelije postaju konveksni poliedri. Dijagram dualan Voronoi-evom dijagramu je dijagram koji nastaje Delaunay-evom triangulacijom [5].



Slika 3.1 Voronoi-ev dijagram

3.2. Delaunay-eva triangulacija

Jedinstvena triangulacija[3] načinjena je tako da ni jedna točka iz skupa *S* ne leži u opisanoj kružnici drugog trokuta (Slika 3.2). Simetrale stranica trokuta određuju opisanu kružnicu trokuta. Proširenjem na 3D prostor trokuti postaju tetraedri i taj se postupak naziva tetraedarizacija [5].



Slika 3.2 Delaunayeva triangulacija

[3] Triangulacija – podjela planarnog objekta na trokute

4. Implementacija

Implementacija simulacije loma čvrstih objekata izrađena je u grafičkom pogonu Unity, a dodatno je korištena otvorena knjižnica MIConvexHull. Unity je odabran iz razloga jer je jednostavan za korištenje, nudi programiranje u C# i sadrži gotove funkcije i komponente (npr. detekcija sudara, čvrsta tijela, fizika). MIConvexHull je knjižnica napisana u C# koja sadrži funkcije za određivanje dijagrama konveksnih tijela, računanje Delanauy-eve triangulacije i Voronoi-evog dijagrama. Ideja simulacije je da se na temelju ulaznih točaka odredi Delaunay-eva triangulacija i da se stvore tetraedri. Svakom tetraedru pridodano je čvrsto tijelo i on je spojen s ostalim tetraedrima koji ga dodiruju te je svakoj toj vezi dodijeljen prag loma.

4.1. Korištenje sustava

Radi lakšeg snaženja u sustavu izrađeno je korisničko sučelje (Slika 4.1.). Na desnoj strani postavljen je panel u kojem se nalaze:

- Gumb „Add Object“ - dodavanje novog objekta, otvara panel za definiranje svojstava objektu kojeg lomimo
- Gumb „Fire“ - otvara panel za definiranje svojstava objektu koji služi kao projektil i kreće se od položaja kamere do položaja na koji miš pokazuje
- Gumb „Add Force“ - otvara panel za definiranje svojstava eksplozivne sile
- Prekidač „Edit“ – ako je uključen odabirom na neki objekt otvara se panel za definiranje svojstava toga objekta
- Prekidač „Delete“ – ako je uključen brišemo odabrane objekte

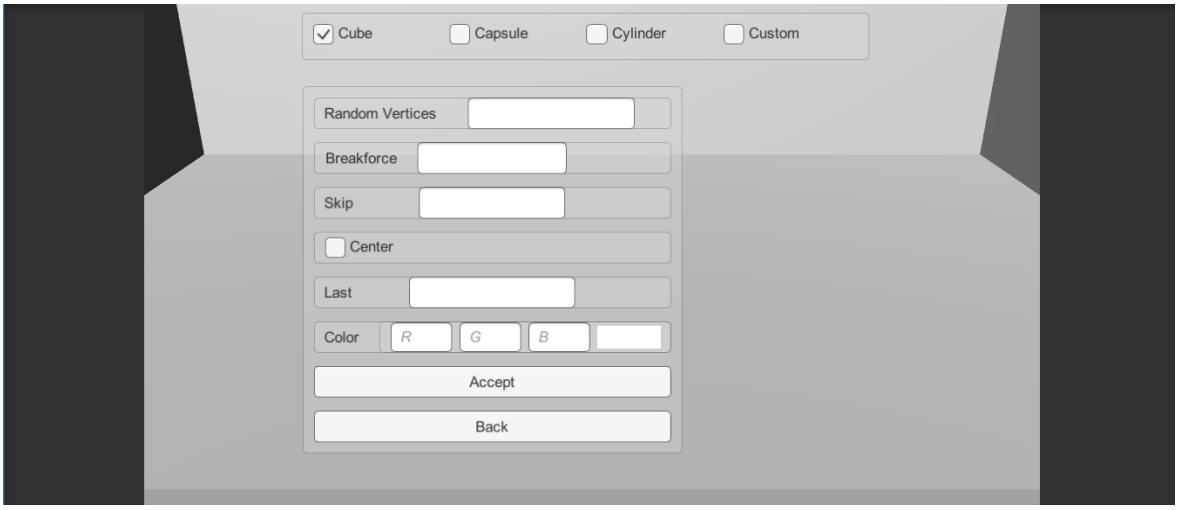
Na lijevoj strani nalazi se panel koji sadrži gumbe:

- „Start“ - služi za pokretanje simulacije
- „Reset“ - zaustavlja simulaciju i vraća objekte na početne položaje
- „Clean“ - briše sve objekte u sceni
- „Pause“ - postavlja stanku u sceni ili pokreće scenu od trenutka kada je postavljena stanka.

Slika 4.1 Početno korisničko sučelje

Pritiskom na gumb „Add Object“ otvara se novi panel s poljima za unos koja predstavljaju pojedino svojstvo objekta (Slika 4.2). Iznad panela nalazi se grupa prekidača koja određuje oblik generiranog tijela, a on može biti kocka, kapsula, valjak ili određeno tijelo koje smo unaprijed zadali („Custom“). Panel sadržava sljedeće elemente:

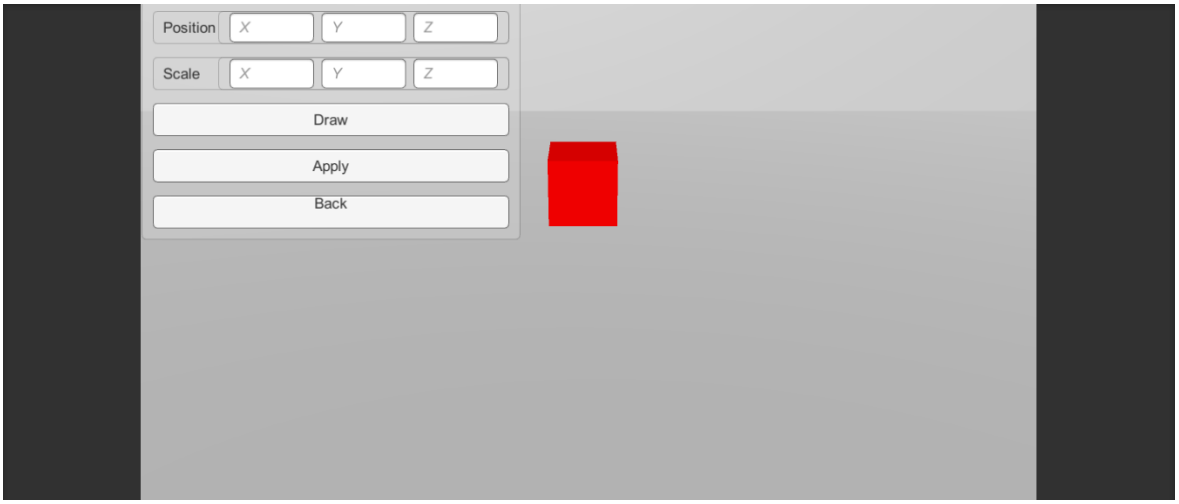
- „Random Vertices“ - broj točaka koje zadajemo, mora biti cijeli broj
- „Breakforce“ - prag loma između veza spojenih tetraedara,
- „Skip“ – broj točaka koji se preskače radi lakše aproksimacije, mora biti cijeli broj
- Prekidač „Center“ – ako je prekidač postavljen onda se dodaje nova točka u sredinu objekta
- „Last“ – koliko sekundi će objekt „živjeti“ nakon pokretanja simulacije prije nego se izbriše iz scene
- „Color“ – tri polja za unos koja predstavljaju redom R, G i B komponentu
- „Accept“ – gumb za prihvat zadanih vrijednosti
- „Back“ – vraćamo se na početni zaslon



Slika 4.2 Prikaz panela za definiranje svojstava objektu kojeg želimo generirati

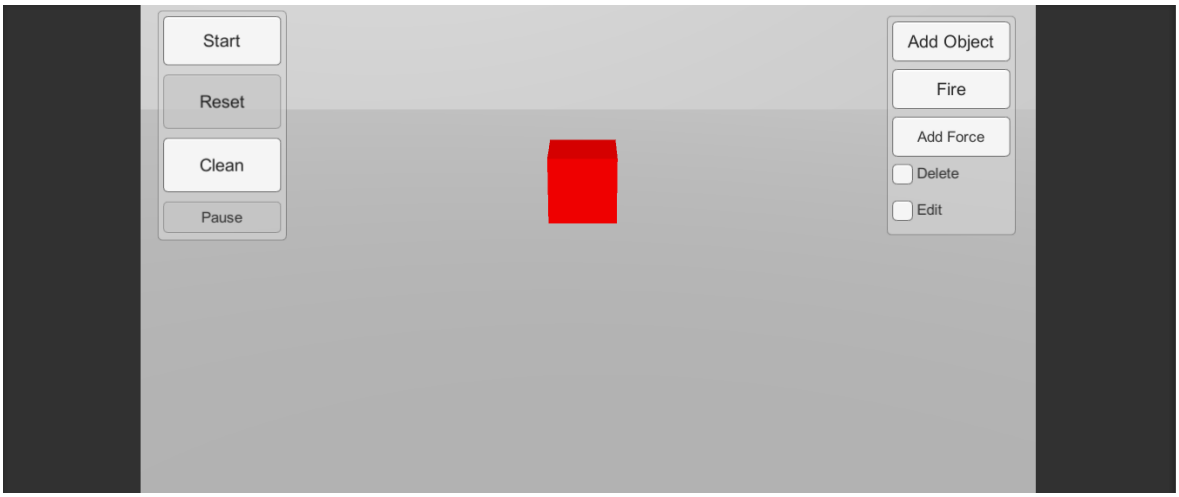
Pritiskom na „Accept“ stvara se novi objekt i otvara se panel za transformiranje stvorenog objekta (Slika 4.3). Inicijalna pozicija će biti (0,10,0), a skaliranje (1,1,1)[4]. Panel sadrži sljedeće elemente:

- „Position“ – 3 polja za unos koja predstavljaju x, y i z pozicije.
- „Scale“ - 3 polja za unos koja predstavljaju x, y i z vrijednosti za skaliranje objekta
- „Draw“ – nanovo crta objekt prema zadanim pozicijama i skaliranjem
- „Apply“ – prihvaća podatke i zatvara panel(Slika 4.4)
- „Back“ – vraćamo se na prijašnji zaslon



Slika 4.3 Primjer stvaranja crvene kocke i panela za unos transformacija

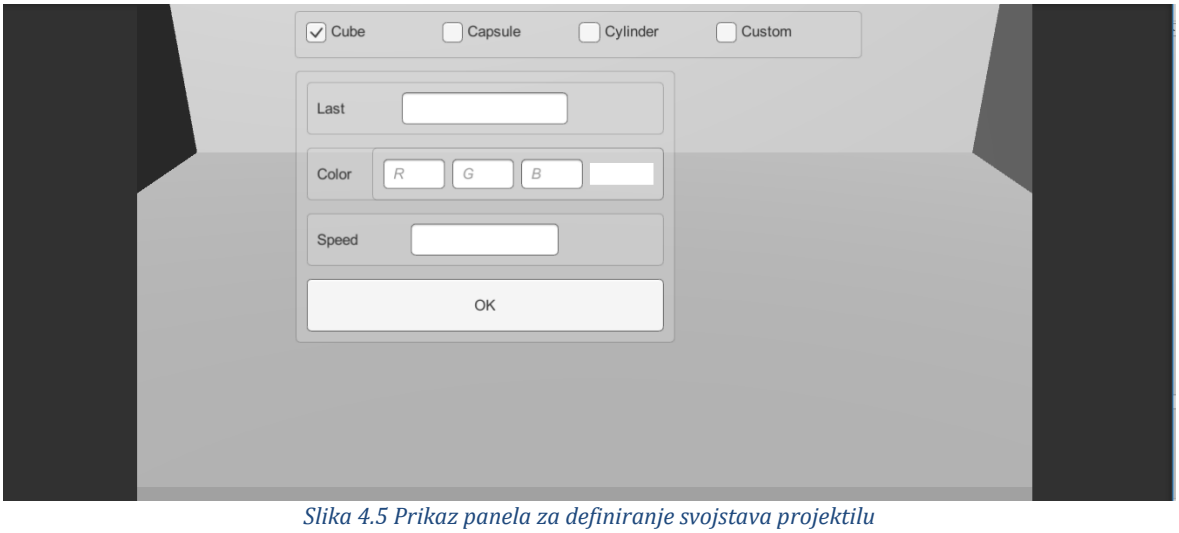
[4] Pozicija i skaliranje je prikazano u obliku vektora (x,y,z)



Slika 4.4 Prikaz tijela i korisnočkog sučelja nakon pritiska tipke „Apply“ u panelu za tranformiranje objekta

Pritiskom gumba „Fire“ otvara se panel za postavljanje svojstava projektilu kojega ispucavamo klikom lijeve tipke miša (Slika 4.5). Iznad panela nalazi se grupa prekidača koja se ponaša na isti način kao i grupa prekidača pri dodavanju novoga objekta samo što se u ovome slučaju odabire oblik projektila. Panel sadrži sljedeće elemente:

- „Last“ – koliko će sekundi projektil „živjeti“ nakon što ga ispucamo
- „Color“– 3 polja za unos R, G i B komponente boje
- „Speed“ – brzina koju će projektil ima kada ga ispucamo iz pozicije kamere
- „OK“ – podaci se prihvaćaju i vraćamo se na prijašnji zaslon



Slika 4.5 Prikaz panela za definiranje svojstava projektilu

Pritiskom gumba „Add Force“ otvara se panel za postavljanje svojstava eksplozivne sile (Slika 4.6). Eksplozivna sila nastaje klikom na lijevu tipku miša tijekom simulacije na mjestu gdje miš pokazuje. Panel sadrži sljedeće elemente:

- „isOn“ – ako je prekidač uključen onda umjesto projektila nastaje eksplozivna sila

tijekom simulacije

- „Radius“ – veličina područja u obliku kugle koju će sila obuhvatiti
- „Power“ – jačina eksplozivne sile
- „Back“ – prihvaćanje vrijednosti i vraćanje na prijašnji zaslon

Slika 4.6 Prikaz panela za definiranje svojstava eksplozivne sile

4.2. Opis sustava

Sustav se sastoji od sljedećih razreda:

- *UIController* – služi za manipuliranje elementima korisničkoga sučelja (otvaranje i zatvaranje pojedinih panela)
- *SimulationController* – upravlja podacima koje korisnik unosi i stvara lomljive objekte
- *ObjectModel* – reprezentacija lomljivog objekta
- *CameraMovement* – upravlja položajem glavne kamere
- *ShootingController* – reprezentacija projektila
- *Tetrahedron* – razred iz knjižnice MIConvexHull kojega je bilo potrebno prilagoditi kako bi knjižnica funkcionirala u Unity-u
- Ostali razredi iz knjižnice MIConvexHull

Zbog jednostavnosti, u nastavku ćemo se fokusirati na razrede *SimulationController*, *ObjectModel* i *Tetrahedron*.

Kao što je već rečeno razred *SimulationController* nam služi za upravljanje podacima koje unosimo i crta lomljivi objekt. Pri svakom crtanju lomljivog objekta pokreće se funkcija *SelectObject* koja postavlja točke unutar objekta na slučajnim

pozicijama i crta lomljivi objekt prema već unaprijed zadanim svojstvima. Detaljniji opis dan je u programskom odsječku 1.

```
/* currentObjModel - članska varijabla tipa ObjectModel koja opisuje lomljivi objekt
                     kojega trenutno uređujemo
   objectDictionary - članska varijabla tipa Dictionary koja predstavlja rječnik
                      lomljivih objekata koje scena sadrži
                    - ključ je pozicija pojedinoga objekta */
    private void SelectObject()
    {
        /* poziva se određena funkcija za generiranje točaka na slučajnim pozicijama
         na temenju odabira prekidača */
        if (cubeToggle.isOn)
        {
            currentObjModel.vertices = SetObjectVertices(cube);
        }
        else if (cylinderToggle.isOn)
        {
            currentObjModel.vertices = SetObjectVertices(cylinder);
        }
        else if (capsuleToggle.isOn)
        {
            currentObjModel.vertices = SetObjectVertices(capsule);
        }
        else
        {
            currentObjModel.vertices = SetObjectVertices(custom);
        }

        /* ako lomljivi objekt već generiran u sceni onda se on briše */
        if (currentObjModel.gameObj != null)
        {
            Destroy(currentObjModel.gameObj);
        }

        /* crta se novi lomljivi objekt */
        currentGameObj = currentObjModel.DrawObject();

        /* ako objekt nije već generiran dodajemo ga u rječnik */
        if (!objectDictionary.ContainsKey(currentGameObj.transform.position))
        {
            objectDictionary.Add(currentGameObj.transform.position,
                                 currentObjModel);
    }
```

Programski odsječak 1 Funkcija SelectObject

Za objekte stvaranje točaka obavlja funkcija *SetObjectVertices* koja kao ulaz prima tip *Gameobject*, a vraća listu stvorenih točaka unutar njega (Programski odsječak 2). Ukoliko zadani objekt sadrži puno točaka, pomoću članske varijable *Skip* možemo preskočiti pojedine točke kako bi ubrzali proces generiranja lomljivog objekta, no u tom slučaju stvoreni objekt neće biti istog oblika kao originalni, već ćemo dobiti aproksimaciju njegovog oblika.

```
private List<Vertex> SetObjectVertices(GameObject gameObj)
{
    //lista u koju ćemo unijeti generirane točke
    List<Vertex> vertices = new List<Vertex>();

    /* dohvaćamo sve točke zadanoga objekta, te točke će se nalaziti na površini
       generiranoga lomljivog objekta */
    Vector3[] meshVertices = gameObj.GetComponent<MeshFilter>()
                                    .sharedMesh.vertices;

    //skaliramo objekt prema zadanim vrijednostima
    ScaleVertices(meshVertices);

    //dohvaćamo središte zadanoga objekta
    Vector3 objCenter = gameObj.GetComponent<Rigidbody>().worldCenterOfMass;

    //dodajemo u listu točaka točke na površini točke zadanoga objekta
    //skip članska varijabla označuje broj točaka koje ćemo preskočiti
    for (int i = 0; i < meshVertices.Count(); i += (skip + 1))
    {
        vertices.Add(new Vertex(meshVertices[i]));
    }

    //generiramo točke na slučajnim pozicijama i dodajemo ih u listu
    /* odabire točku na poziciji između slučajno odabrane točke na površini i
       središta */
    for(int i = 0; i< verticesNumber; i++)
    {
        int index = UnityEngine.Random.Range(0, vertices.Count);
        Vector3 point = vertices[index].ToPoint3D();
        float distance = UnityEngine.Random.Range(0.0f, Vector3.Distance(point,
                                                  objCenter));

        Vector3 direction = objCenter - point;
        direction = direction + direction.normalized * distance;

        Vector3 targetPosition = point + direction;

        vertices.Add(new Vertex(targetPosition));

    }

    if (center)
    {
        vertices.Add(new Vertex(objCenter));
    }

    return vertices;
}
```

Programski odsječak 2 Funkcija SetObjectVertices

Unutar razreda *ObjectModel* nalazi se funkcija *DrawObject* koja crta tetraedre prema zadanim točkama tipa *Vertex* i povezuje one koje se dodiruju (Programski odsječak 3). MIConvexHull pruža statičku klasu *Triangulation* koja sadrži funkciju *CreateDelaunay*. U osnovi *CreateDelaunay* kao ulaz prima listu točaka tipa *Vertex*, a vraća listu tetraedara. U prvoj petlji stvaramo sve objekte tipa *GameObject* i

pridružujemo mu određena svojstva, a u drugoj petlji svakog tetraedra spajamo sa susjednim tetraedrima uz pomoć *Joint* komponente. *Joint-u* definiramo prag lomljenja („breakforce“) i čvrsto tijelo („RigidBody“) koje je spojeno s njim.

```
public GameObject DrawObject()
{
    /* stvaramo tetraedre preko statičke funkcije CreateDelaunay koja kao
       ulaz prima listu točaka koje smo unaprijed zadali i na temelju toga
       vraća listu tetraedara */
    List<Tetrahedron> tetrahedrons = Triangulation.CreateDelaunay<Vertex,
                                     Tetrahedron>(vertices).Cells.ToList();

    Dictionary<Vector3, GameObject> tetraDict = new Dictionary<Vector3,
                                                GameObject>();

    GameObject parent = new GameObject();
    parent.transform.position = pos;
    GameObject[] tetraList = new GameObject[tetrahedrons.Count];

    // za svaki tetraedar u listi stvori GameObject u sceni
    for(int i = 0; i < tetrahedrons.Count; i++)
    {
        GameObject go = tetrahedrons[i].CreateMesh();
        go.transform.parent = parent.transform;
        go.transform.position = parent.transform.position;
        go.GetComponent<Renderer>().material = mat;
        tetraDict.Add(tetrahedrons[i].GetCenter(), go);
        tetraList[i] = go;
    }
    gameObj = parent;

    //sljedećom petljom povezuju se Joint-ima svi tetraedri koji se dodiruju
    for (int i = 0; i < tetrahedrons.Count; i++)
    {
        foreach (var t in tetrahedrons[i].Adjacency)
        {
            GameObject tetraGo = tetraList[i];
            if (t != null && tetraGo!= null)
            {
                GameObject adjTetra;
                FixedJoint joint = tetraGo.AddComponent<FixedJoint>();
                if (tetraDict.TryGetValue(t.GetCenter(), out adjTetra) &&
                    joint.connectedBody == null)
                {
                    joint.connectedBody = adjTetra.GetComponent<Rigidbody>();
                    joint.breakForce = breakforce;
                }
            }
        }
    }
    gameObj.tag = objectTag;
    return gameObj;
}
```

Programski odsječak 3 Funkcija DrawObject

Funkcija *CreateMesh* koja se nalazi u razredu *Tetrahedron* stvara poligone tetreadara i vraća *GameObject* kojemu su pridruženi ti poligoni (Programski odsječak

4). Način na koji Unity crta objekte je da komponenti *Mesh* pridoda niz točaka[5] i indeksa[6]. Svaka 3 indeksa u nizu predstavljaju jedan poligon odnosno trokut na temelju niza zadanih točaka 3D geometrijskog objekta. *Mesh* dodamo *GameObject*-u koji treba sadržavati komponentu *MeshFilter* koja sprema i *MeshRenderer* koja prikazuje *Mesh* i Unity će ga zatim nacrtati u sceni. Način na koji stvaramo novi tetraedar je da se prvo postavi lista cijelih brojeva koji sežu redom od 0 do 11. Time se postave indeksi na točke. Zatim se dodaje 12 točaka, pazeći na redoslijed, preko *MakeFace* funkcije. *MakeFace* funkcija dodaje u listu točaka po tri nove točke. Kao ulaz prima indekse točaka tetraedara u nizu (što znači da će ti indeksi biti brojevi od 0 do 3), središte i listu točaka koju „puni". Logično bi bilo u listu točaka dodati samo 4 točke i u tom slučaju će lista indeksa sadržavati 12 cijelih brojeva čija je vrijednost od 0 do 3, no problem nastaje pri određivanju normala. *Mesh* sadrži i niz normala te Unity automatski postavlja da je njegova veličina jednaka veličini niza točaka. Ako bi postojale četiri točke u nizu onda bi postojale i četiri normale i tada bi nam sjenčanje bilo potpuno neispravno te je iz tog razloga postavljeno 12 točaka. Preko funkcija *SetVertices* i *SetTriangles* dodajemo liste točaka i indeksa u *Mesh*. Drugi način bi bio pretvoriti listu u niz i dodati ga *Meshu* kao u programsko odsječku 5, no na taj način gubimo na performansama zbog konverzije liste u niz, a funkcije *SetVertices* i *SetTriangles* efikasno i brzo čitaju podatke iz liste.

```
public GameObject CreateMesh()
{
    //točke tetraedara
    var points = new List<Vector3>(Enumerable.Range(0, 4).Select(i =>
                                    GetPosition(i)));

    // center = suma(p[i]) / 4
    var center = points.Aggregate(new Vector3(), (a, c) => a + c) /
                                   (float)points.Count;
    this.center = center;

    /* lista koja određuje indekse trokuta koji će biti povezani, lista
       sadrži brojeve od 0 do 11 */
    var indices = new List<int>(Enumerable.Range(0, 12));
    //lista točaka
    var vertices = new List<Vector3>();


    /* MakeFace funkcija puni listu vertices s tri točke koje
       predstavljaju jedan poligon */
    MakeFace(0, 1, 2, center, vertices);
```

[5] Niz cijelih brojeva
[6] Niz tipa *Vector3*

```
            MakeFace(0, 1, 3, center, vertices);
            MakeFace(0, 2, 3, center, vertices);
            MakeFace(1, 2, 3, center, vertices);

            // postavljamo točke i indekse geometrijskom objektu
            Mesh mesh = new Mesh();
            mesh.SetVertices(vertices);
            mesh.SetTriangles(indices, 0);

            GameObject go = new GameObject();
            go.AddComponent<MeshFilter>();
            go.AddComponent<MeshRenderer>();
            go.AddComponent<MeshCollider>().sharedMesh = mesh;
            go.GetComponent<MeshCollider>().convex = true;

            // postavljamo geometrijski objekt GameObject-u
            go.GetComponent<MeshFilter>().mesh = mesh;

            Rigidbody rb = go.AddComponent<Rigidbody>();
            rb.useGravity = true;
            rb.isKinematic = true;
            go.GetComponent<MeshFilter>().mesh.RecalculateNormals();
            return go;
        }


        void MakeFace(int i, int j, int k, Vector3 center,
                      List<Vector3> vertices)
        {
            var u = GetPosition(j) - GetPosition(i);
            var v = GetPosition(k) - GetPosition(j);

            // računa normalu i ravninu
            var n = Vector3.Cross(u, v);
            var d = -Vector3.Dot(n, center);

            // provjerava se je li centar iznad ravnine
            var t = Vector3.Dot(n, (Vector3)GetPosition(i)) + d;
            if (t >= 0)
            {
                // indeksi su točnom redoslijedu
                vertices.Add(GetPosition(i));
                vertices.Add(GetPosition(j));
                vertices.Add(GetPosition(k));
            }
            else
            {
                // mijenjamo i i k indekse kako bi dobili ispravne normale
                vertices.Add(GetPosition(k));
                vertices.Add(GetPosition(j));
                vertices.Add(GetPosition(i));
            }
        }
```

Programski odsječak 4 Funkcije MakeFace i CreateMesh

```
                    mesh.vertices = vertices.ToArray();
                    mesh.triangles = indices.ToArray();
```

Programski odsječak 5 Neispravan način dodavanja liste točaka i indeksa u objekt tipa Mesh

5. Testiranje

U ovome poglavlju opisano je ponašanje sustava kroz ekperimentalna mjerenja i ograničenja s obzirom na prijenosno računalo[7] na kojem je on izrađen i testiran.

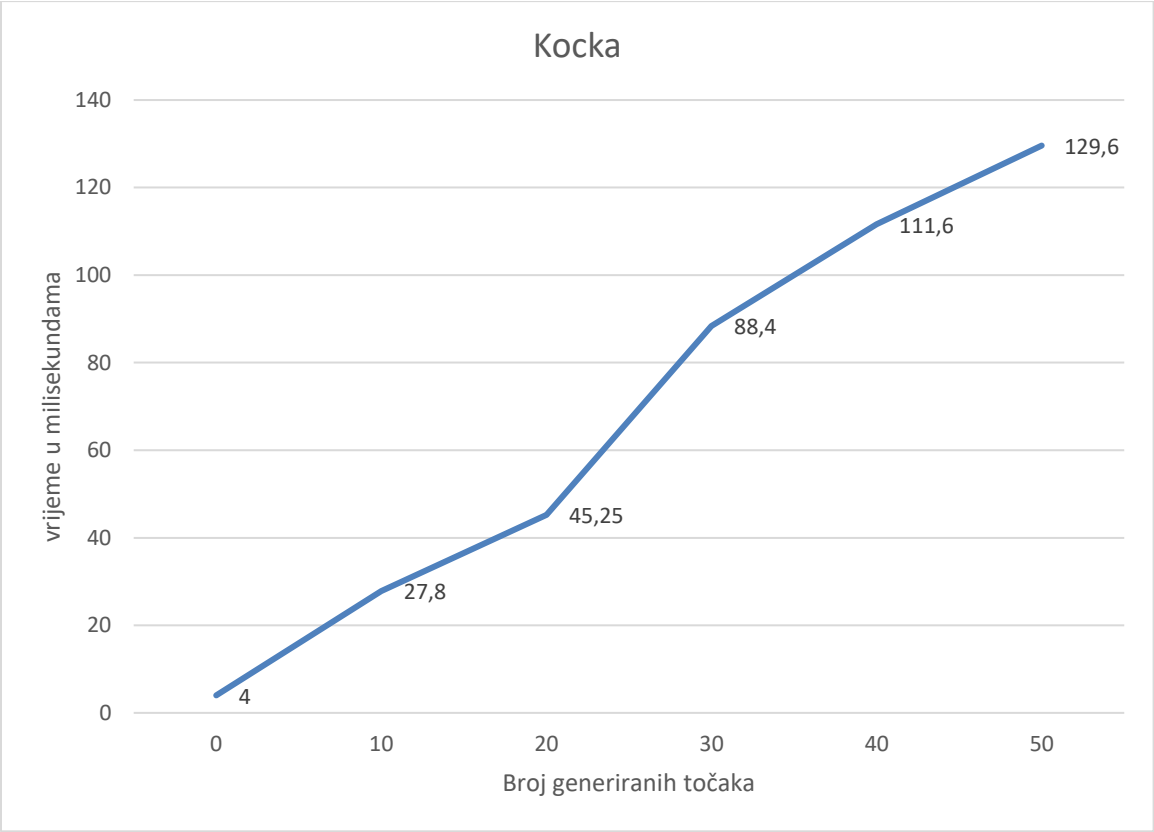
5.1. Mjerenja

Kako bi se provjerilo koliki je vremenski interval potreban da se stvori lomljivi objekt u ovisnosti o broju slučajno generiranih točaka, izvršavanje funkcije *SetSelectedObject* se mjerilo virtualnom štopericom u milisekundama, a primjer koda prikazan je u programskom odsječku 6. Mjerenje se obavilo na modelu kocke (24 točke, 8 poligona), kapsule (550 točke, 832 poligona), valjka (88 točke, 80 poligona) i ptice (958 točke, 1394 poligona) tako što se inkrementalno povećavao broj slučajno generiranih točaka s korakom od 10 točaka. Radi bolje preciznosti računala se srednja vrijednost 5 mjerenja vremenskog izvršavanja funkcije *SetSelectedObject* za određeni broj točaka. Za jednostavne modele kocke i valjka vremenski interval potreban da se lomljivi objekt stvori u linearnoj je ovisnosti o broju generiranih točaka (Graf 1 i 3), no za složenije modele to neće vrijediti. Grafovi 2 i 4 prikazuju kako će interval prvo padati, ali će nakon određenoga broja točaka njegova vrijednost početi rasti što znači da vremenska složenost funkcije koja obavlja triangulaciju ne ovisi nužno o broju točaka te će se ona brže izvršiti kod složenijih modela ako unutar njega dodamo točke. Pretpostavlja se da je uzrok opisanome ponašanju sustava implementacija Delaunay-ove triangulacije.

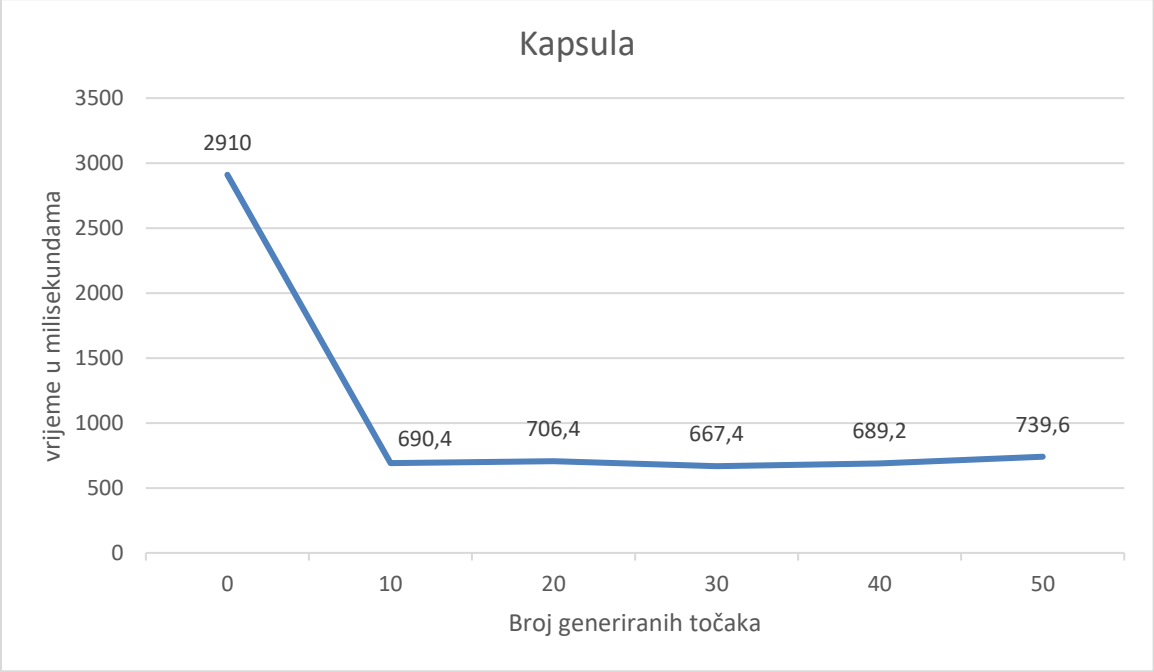
```
var watch = System.Diagnostics.Stopwatch.StartNew();

//funkcija koja se mjeri

watch.Stop();
var elapsedMs = watch.ElapsedMilliseconds;
Debug.Log(elapsedMs);
```

Programski odsječak 6

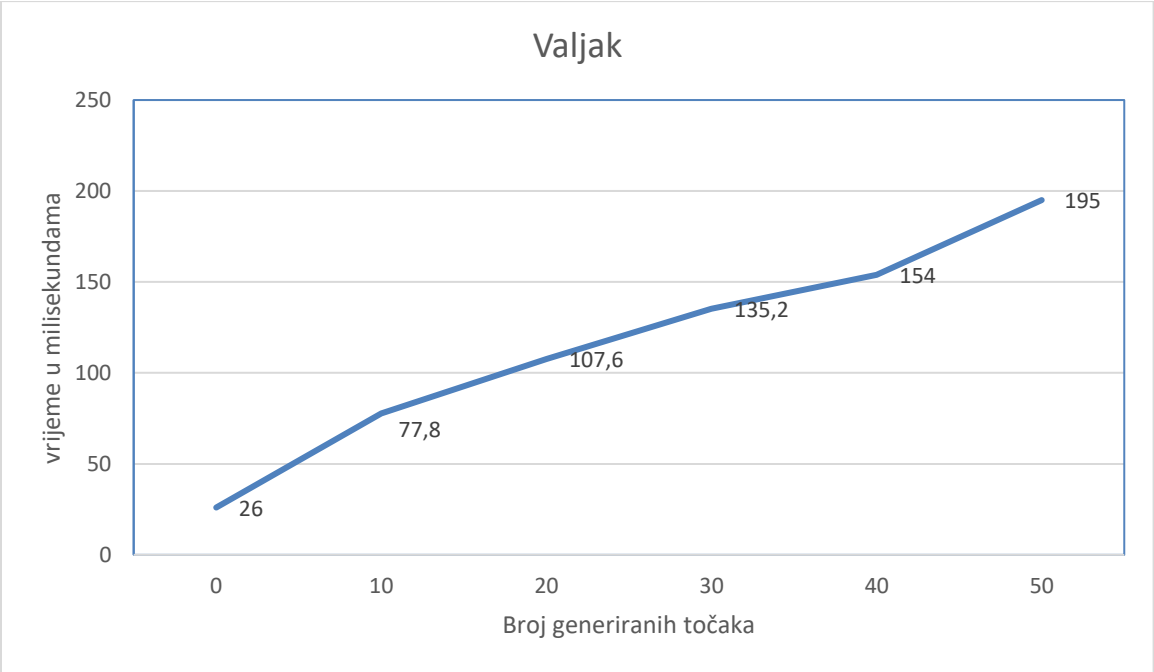
[7] SATELLITE C55-C-1J6 - brzina procesora : 2.00 GHz; RAM : 4,096 MB; Grafička kartica: NVIDIA® GeForce® 920M with NVIDIA® Optimus™ Technology, 1GB dedicated VRAM



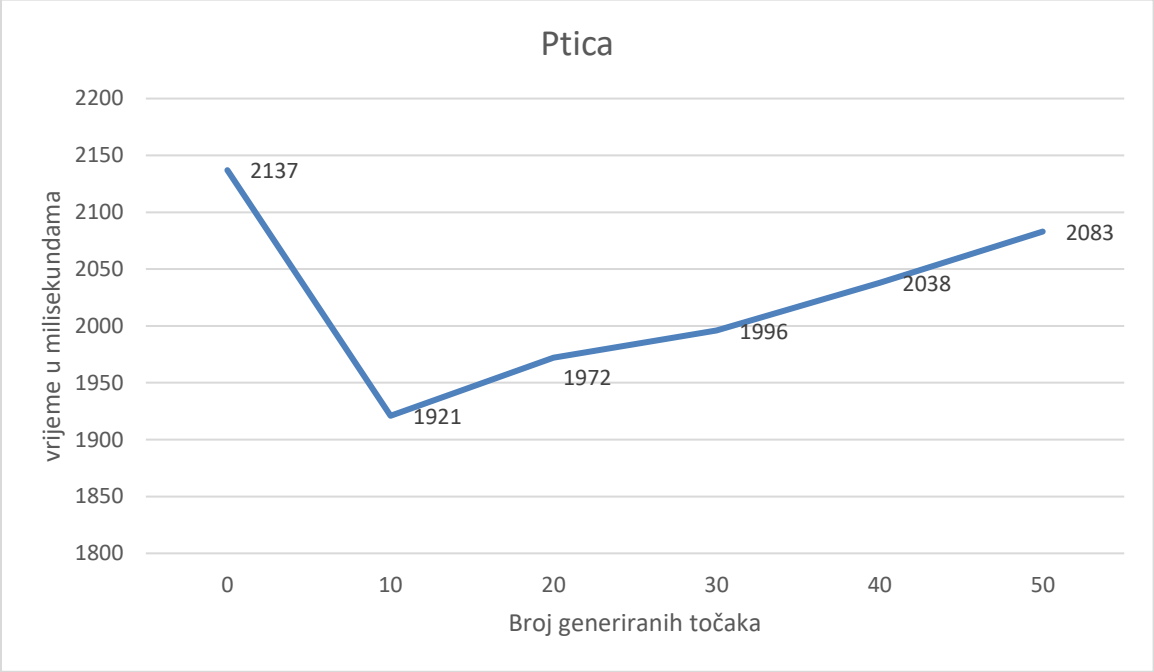
Graf 1 Utjecaj generiranih točaka na vrijeme potrebno da se lomljivi model kocke generira



Graf 2 Utjecaj generiranih točaka na vrijeme potrebno da se lomljivi model kapsule generira



Graf 3 Utjecaj generiranih točaka na vrijeme potrebno da se lomljivi model valjka generira

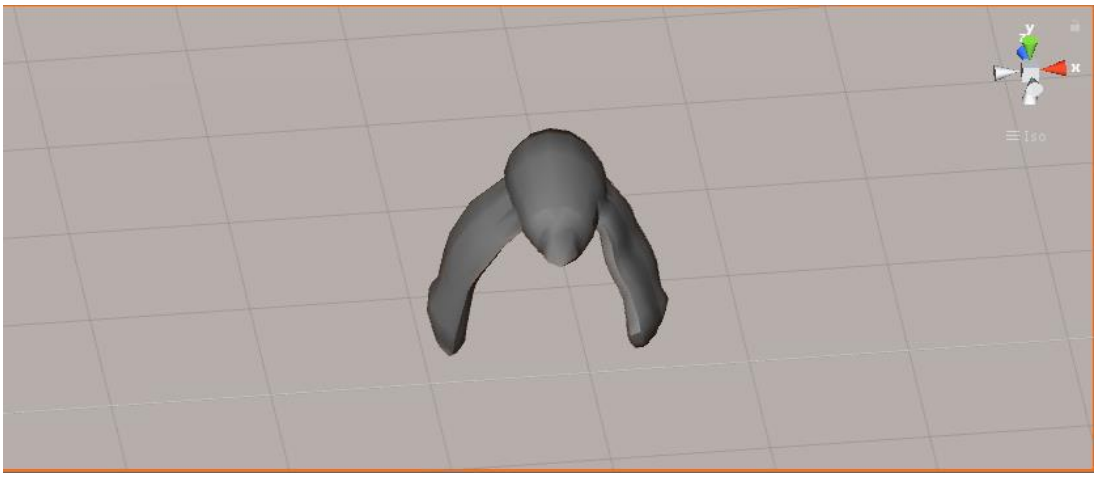


Graf 4 Utjecaj generiranih točaka na vrijeme potrebno da se lomljivi model ptice generira

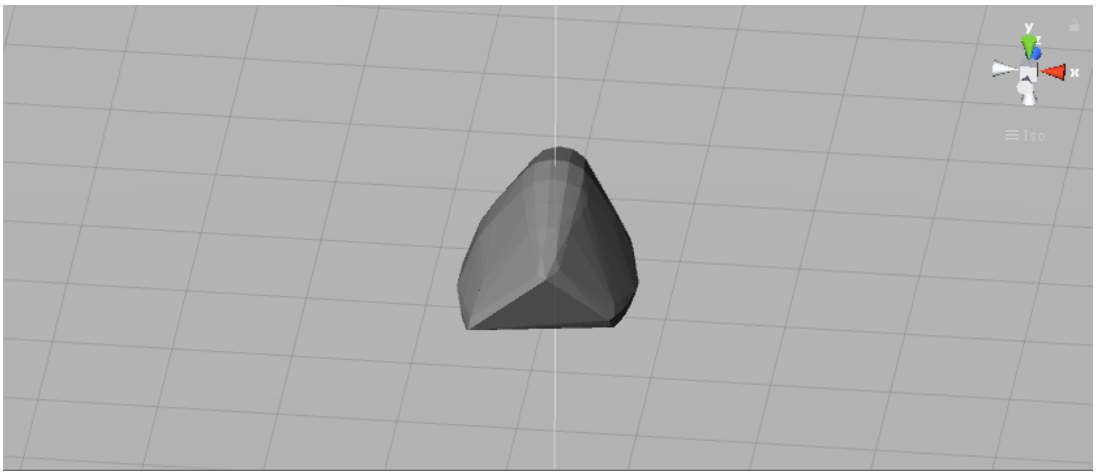
5.2. Ograničenja

Ukoliko u sustavu pokušamo slomiti konkavan objekt, to nam neće poći za rukom. Problem nastaje pri generiranju točaka jer će se generirati točke koje su između slučajno odabrane točke i središta i na taj način lomljivi objekt koji se stvara će uvijek biti konveksan. Slike 4.1 i 4.2 prikazuju primjer generiranja lomljivoga

objekta na temelju 3D modela ptice. Problem bi se mogao riješiti tako da objekt podijelimo na konveksne dijelove i zatim izračunamo točke.



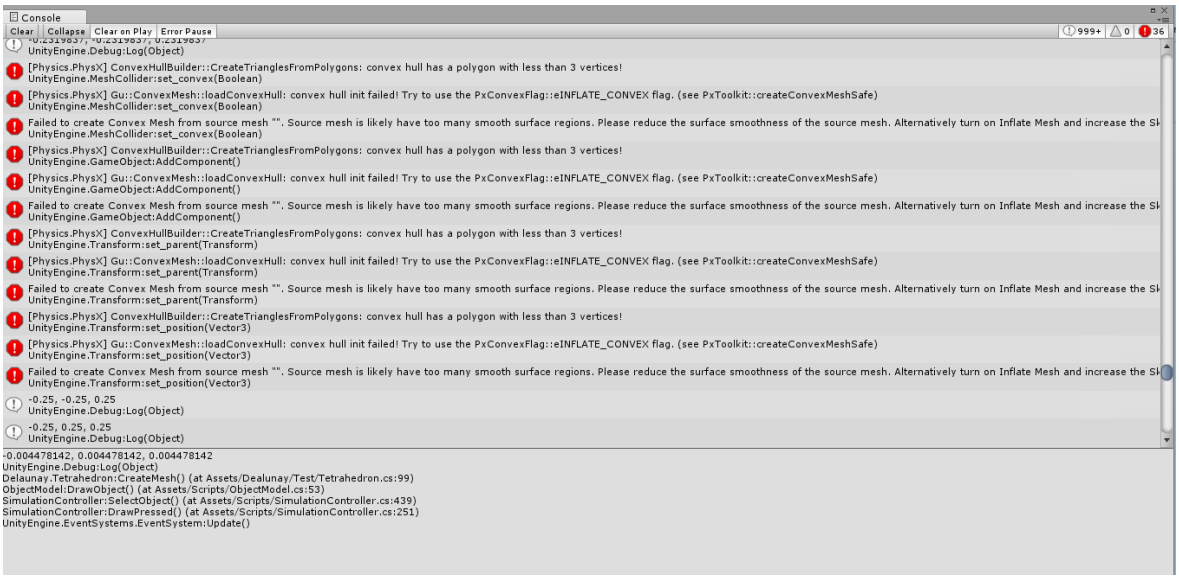
Slika 5.1 3D model ptice



Slika 5.2 Lomljivi objekt generiran na temelju modela ptice

U nekim slučajevima pri stvaranju lomljivoga objekta Unity generira iznimku koja označuje da se komponenti *MeshCollider* ne postavlja ispravan 3D geometrijski oblik (Slika 4.3). Ta iznimka se ne može uhvatiti u C# jer je Unity generira iz C++[8] te je dosta teško odrediti razlog nastanka. Unatoč tome, simulacija se odvija bez problema pa ni posljedice iznimke nisu sasvim jasne. Pretpostavka je da iznimka nastaje zbog numeričkih pogrešaka jer je njena vjerojatnost pojavljivanja veća što je objekt manji ili što je veći broj točaka koje se generiraju unutar objekta.

[8] U Unity-u je fizika napisana u C++



Slika 5.3 Prikaz iznimki koje se generiraju tijekom stvaranja lomljivog objekta

Kod složenijih modela postoji velika vjerojatnost preopterećenja sustava. Kako bi izbjegli tu situaciju poželjno je postaviti varijablu *Skip* kako bi se modeli pojednostavili ili pripaziti na zadavanje broja slučajno generiranih točaka.

6. Zaključak

Računalna grafika bez lomljenja i uništavanja objekata bi bila nezamisliva. Simulacije fizike tada ne bi bile ispravne, a videoigre bi nam bile nezanimljive. Računalni svijet bez lomljenja i uništavanja bi bio vrlo dosadan, no problem nastaje kada fiziku iz stvarnoga svijeta pokušamo izravno preslikati u virtualni sustav. Stvarni svijet djeluje na razini molekula i atoma, a takav sustav je vrlo teško preslikati i implementirati u računalni sustav. Iz tog razloga ostvarene su metode koje ne izvode realnu fiziku, već se pokušava predočiti izgled lomljenja čvrstog tijela. Unatoč tome, tim metodama se dobiva vrlo dobra aproksimacija realnoga svijeta. U radu je opisan takav sustav koji se temelji na Delaunay-evoj triangulaciji i međusobnom spajanju čvrstih tijela zajedno sa svim njegovim manama i prednostima. Daljnji plan je unaprijediti sustav, dodati mogućnost kretnje lomljivog objekta, pokušati implementirati ostale metode lomljenja (npr. boolean, metoda konačnih elemenata), a zatim pokušati takav sustav iskoristiti za neku jednostavnu videoigru.

LITERATURA

[1] M. West, »Shattering-reality,« 5 Siječanj 2007. [Mrežno]. Available: http://cowboyprogramming.com/2007/01/05/shattering-reality/.

[2] E. Coumans, »Opinion: Destruction,« 6 Rujan 2011. [Mrežno]. Available: http://www.gamasutra.com/view/news/126940/Opinion_Destruction.php.

[3] R. Hettich, Approaches to destruction effects in real-time computer graphics, Završni rad: Fakultät für Informatik und Wirtschaftsinformatik, 2013.

[4] M. Vadlja, Modeliranje i simulacija deformabilnih objekata, Zagreb: FER, 2011.

[5] M. Müller, L. McMillan, J. Dorsey i R. Jagnow, »Real-Time Simulation of Deformation and,« *Computer Animation and Simulation,* 2001.

[6] »Detekcija sudara,« u *Računalna grafika*, Zagreb, FER, pp. 15-18.

[7] M. Čupić i Ž. Mihajlović, Interaktivna računalna grafika kroz primjere u OpenGL-u, 2016.

Simulacija lomljenja objekata

Sažetak

Ovaj rad obrađuje simulaciju lomljenja objekata. Opisane su metode koje se u današnje vrijeme najčešće koriste odnosno prezentirani su načini pripreme 3D objekta i postupci njegova lomljenja u stvarnom vremenu, a izvedena je i matematička pozadina Delanauy-ove triangulacije pošto smo se njome bavili tijekom izrade sustava za simulaciju. Zatim je predstavljena izvedba i korištenje sustava za lomljenje tijela kao i opis njegova ponašanja kroz analizu mjerenja. Na kraju su pojašnjene mane i ograničenja sustava kako bi se u daljnjem razvoju te mane ispravile.

Ključne riječi: lomljenje objekata, triangulacija, simulacija u stvarnom vremenu, sustav za simulaciju fizike, pucanje, Unity, C#

Breaking Objects Simulation

Abstract

This paper describes the simulation of breaking objects. We present basic methods for preparing and breaking 3D objects in real-time that are nowadays in use. Mathematical background of Delaunay triangulation is explained because of her use in the simulation implementation. Then, the behaviour and performance of the system for simulation is presented through experimental analysis. Cons and limits to the simulation are explained in the last chapter so that they would be corrected in further development.

Keywords: object breaking, triangulation, real-time simulation, physics simulation, fracture, Unity, C#