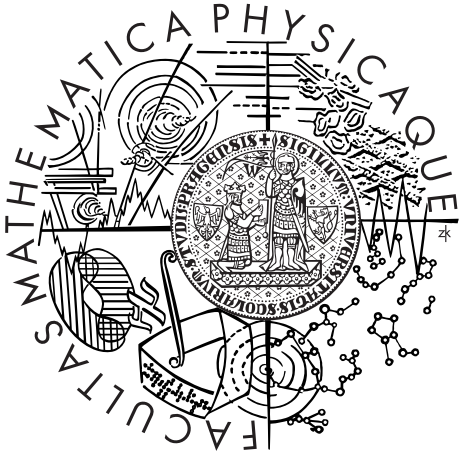
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Marek Beňovič

Podpora fraktúr pre jbox2d engine

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: RNDr. Jan Kofroň, Ph.D.
Studijní program: Informatika
Studijní obor: Programování

Praha 2015

Rád by som poďakoval vedúcemu mojej práce RNDr. Jánovi Kofroňovi, Ph.D. za jeho ochotu a cenné rady. Taktiež ďakujem mojej rodine a priateľom za podporu počas štúdia.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Podpora fraktúr pre jbox2d engine

Autor: Marek Beňovič

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: RNDr. Jan Kofroň, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: JBox2D je herní engine simulující fyziku pevných těles a kapalin v 2D prostoru. Práce poskytuje rozšíření knihovny JBox2D umožňující tříštění těles po jejich vzájemné kolizi. Důraz je kladen na plynulost činnosti algoritmu v reálném čase, nízké nároky na výkon procesoru a přirozenost průběhu procesů tříštění. Algoritmus také poskytuje možnost definovat materiály těles a nastavovat jejich vlastnosti, na nichž závisí průběh simulace tříštění těchto těles. Je k dispozici jednoduché programátorské rozhraní založené na logice knihovny. Na demonstraci funkčnosti daného řešení práce obsahuje i jednoduchý framework s testovacími scénáři napodobujícími fraktury objektů. Práce poskytuje nové možnosti při vývoji 2D her pro mobilní zařízení a osobní počítače.

Klíčová slova: JBox2D, fyzikální engine, fyzika projektilu, voroného fragmentace, konvexní dekompozice

Title: Support for fractures in jbox2d engine

Author: Marek Beňovič

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Jan Kofroň, Ph.D., Department of Distributed and Dependable Systems

Abstract: JBox2D is a game engine simulating the physics of solid objects and liquids in a 2D space. This project provides a JBox2D library extension that allows for fracturing of objects after their collision. The presented algorithm prioritizes its smooth running in real time, low processing power requirements and a natural flow of the fracturing processes. The algorithm also provides a possibility to define the materials of the objects to be fractured and set their properties, which in turn determine the outcome of the simulation process of fracturing these objects. A simple programming interface based on the logic of the library is provided. In order to demonstrate the usability of the solution, the project also contains a simple framework with test scenarios simulating fracturing of objects. This project provides new possibilities for developing 2D games for mobile devices and personal computers.

Keywords: JBox2D, physics engine, bullet physics, voronoi shattering, convex decomposition

Názov práce: Podpora fraktúr pre jbox2d engine

Autor: Marek Beňovič

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedúci bakalárskej práce: RNDr. Jan Kofroň, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: JBox2D je herný engine simulujúci fyziku pevných telies a kvapalín v 2D priestore. Práca poskytuje rozšírenie knižnice JBox2D umožňujúce trieštenie telies po ich vzájomnej kolízii. Dôraz je kladený na plynulosť činnosti algoritmu v reálnom čase, nízke nároky na výkon procesora a prirodzenosť priebehu procesov trieštenia. Algoritmus taktiež poskytuje možnosť definovať materiály telies a nastavovať ich vlastnosti, od ktorých závisí priebeh simulácie trieštenia týchto telies. Je k dispozícii jednoduché programátorské rozhranie založené na logike knižnice. Na demonštráciu funkčnosti daného riešenia práca obsahuje aj jednoduchý framework s testovacími scenármi simulujúcimi fraktúry objektov. Práca poskytuje nové možnosti pri vývoji 2D hier pre mobilné zariadenia a osobné počítače.

Kľúčové slová: JBox2D, fyzikálny engine, fyzika projektilu, voroného fragmentácia, konvexná dekompozícia

Obsah

1. Úvod

1.1 Motivácia

Vývoj 2D počítačových hier je neodmysliteľnou súčasťou informatiky. Pri vývoji mnohých hier je nutné nadefinovať pravidlá pre riešenie interakcí medzi objektmi. V hrách, ktoré sa snažia simulovať reálne správanie telies, sa na tento účel používa *fyzikálny engine*. Ten umožňuje určiť základné fyzikálne vlastnosti objektom, sústavám objektov a scéne a na základe nich dokáže pre konkrétne časy poskytnúť informácie o ich stave. Vzhľadom na zložitosť problému sa simulácia posúva v diskrétnych časových jednotkách.

V súčasnosti existuje viacero enginov určených na simuláciu fyziky pevných telies v 2D ako napr. *Box2D*, *Chipmunk* alebo *Unity*. Pri výbere enginu je potrebné zohľadňovať parametre ako:

- API
- funkcionalita
- komunita programátorov
- dokumentácia
- podpora pre konkrétny programovací jazyk.

Rozsiahla funkcionalita, dobrá dokumentácia, podpora Javy a široká komunita aktívnych vývojárov viedla k výberu enginu Box2D.

Box2D [1] je open source C++ engine simulujúci fyziku pevných telies v 2D. Je vyvinutý *Erinom Catoonom* a distribuovaný pod licenciou *Zlib* [9]. Engine je kombinovaný s knižnicou *Liquid fun*, ktorá rozširuje Box2D o podporu simulácie kvapalín. *JBox2D* [2] je Java port pre Box2D vedený *Danielom Murphym* a v roku 2007 distribuovaný pod rovnakou licenciou. Okrem simulácie pevných telies poskytuje širokú škálu funkcií, ako nastavenie gravitácie, simuláciu spojov rôznych typov, serializáciu, detekciu spojitých kolízií a iné. To, čo však tomuto, ako aj iným enginom chýba, je podpora fragmentácie telies.

1.2 Cieľ práce

Práca si kladie za cieľ implementovať algoritmus umožňujúci rozpad kolidujúcich telies. Cieľom je simulácia procesov trieštenia objektov, ktorá bude schopná bežať v reálnom čase, bude vykazovať nízke výkonnostné nároky a dávať výsledky javiace sa do čo najväčšej možnej miery prirodzene. Dôraz je tiež kladený na jednoduchosť programátorského rozhrania. Účelom je tiež simulovanie rôznych typov materiálov s rozličnými vlastnosťami a ich následná vizualizácia. Práca je primárne určená pre podporu vývoja hier.

1.3 Štruktúra práce

Teoretická časť práce je venovaná popisu princípu geometrických algoritmov, ich vlastnostiam a časovej zložitosti. Súčasťou práce je tiež popis implementácie a vizualizácia testovacích scenárov.

V nasledujúcej kapitole je popísaná práca s knižnicou JBox2D, základné rozhranie a prezentovaný postup, ako vytvoriť jednoduchú simuláciu, ktorá ponúka základnú predstavu o fungovaní fyzikálnych enginov. Tretia kapitola uvádza do problematiky teórie grafov a venuje sa riešeniu výpočtu voroného diagramu. V štvrtej kapitole je priblížený problém konvexnej dekompozície. Piata kapitola prezentuje spôsob, ako pomocou voroného diagramu a parametrov kolízie fragmentovať teleso. Šiesta kapitola sa venuje implementácii popísaných algoritmov a ich aplikovaniu na JBox2D engine. V siedmej kapitole sú popísané spôsoby definície materiálu, ich vlastnosti a sú prezentované výsledky simulácií. Ôsma kapitola je venovaná programátorskému rozhraniu. V závere sú uvedené možnosti, ako v práci pokračovať a aplikovať ju do praxe.

2. JBox2D engine

JBox2D engine je Java port fyzikálneho enginu Box2d. Knižnica je voľne dostupná na stránkach *GitHub*-u [2].

2.1 Štruktura kódu a jeho rozhranie

Engine je implementovaný v balíku $org/jbox2d$. Implementácia fragmentácie telies sa nachádza v priečinku $org/jbox2d/fracture$. Z dôvodu zakomponovania rozšírenia bolo nutné v pôvodnom engine vykonať malé zmeny v implementácii, ktorá okrem iného rozširuje rozhranie o nové premenné a metódy.

Rozhranie knižnice je pomerne obsiahle, preto bude v tejto sekcii priblížená len tá kľučová časť, ktorá umožní programátorovi vytvoriť jednoduchý testovací scenár a poskytne základný prehľad o fungovaní knižnice. Prezentované je rozhranie pôvodnej verzie knižnice a pridaná funkcionalita bude popísaná až v ďalších kapitolách.

- `dynamics.World` Hlavný objekt reprezentujúci svet (simulačné prostredie), do ktorého budú následne umiestňované objekty. Dôležité funkcie:
  - `Body createBody()` Vytvorí teleso.
  - `void setParticleRadius()` Nastaví polomer čiastočiek kvapalín.
  - `Joint createJoint()` Vytvorí spoj.
  - `void destroyBody()` Zmaže teleso.
  - `Joint destroyJoint()` Zruší spoj.
  - `void setParticleDensity()` Nastaví hustotu kvapalín.
  - `ParticleGroup createParticleGroup()` Vloží kvapalinu do simulácie do nadefinovaného útvaru.
- `dynamics.Body` Reprezentuje pevné teleso, ktoré je súčasťou sveta. Má nasledovné atribúty:
  - `float m_angularVelocity` Uhlová rýchlosť (rad/s).
  - `Contact m_contactList` Spojový zoznam kontaktov.
  - `int m_fixtureCount` Počet predmetov telesa.
  - `Fixture m_fixtureList` Spojový zoznam predmetov.
  - `Vec2 m_linearVelocity` Vektor pohybu ťažiska.
  - `float m_mass` Hmotnosť.
  - `BodyType m_type` Typ.
  - `Transform m_xf` Aktuálna transformácia telesa.
  - `Transform m_xf0` Transformácia pred začatím frame-u.
  - `Fixture createFixture()` Vytvorí predmet.
- `dynamics.BodyDef` Šablóna definujúca atribúty telesa.
  - `float angle` Uhol otočenia.

  - **`float angularVelocity`** Rýchlosť otáčania.
  - **`boolean fixedRotation`** Povolenie rotácie.
  - **`Vec2 linearVelocity`** Vektor pohybu.
  - **`Vec2 position`** Pozícia.
  - **`BodyType type`** Typ.

- **`dynamics.BodyType`** Enum trieda určujúca typ telesa. V simulácii sa budú používať dva typy: *statické* a *dynamické*. Teleso dynamického typu je schopné pod vplyvom vonkajších síl (gravitácia, kolízia) meniť svoju polohu, zatiaľ čo statické teleso je fixne umiestnené na jednom mieste a svoju polohu nemení.

- **`dynamics.Fixture`** Predmet. Je používaný na definovanie častí telesa, z ktorých je teleso vyskladané. Transformáciu dedí od rodičovského telesa.

  - **`Body m_body`** Rodičovské teleso.
  - **`float m_density`** Hustota.
  - **`float m_friction`** Trenie.
  - **`float m_restitution`** Odrazivosť.
  - **`Shape m_shape`** Tvar.

- **`dynamics.FixtureDef`** Šablóna definujúca predmet.

  - **`float density`** Hustota.
  - **`float friction`** Trenie.
  - **`float restitution`** Odrazivosť.
  - **`Shape shape`** Tvar.

- **`dynamics.joints`** Balík obsahujúci spoje. Tie definujú silové pôsobenie medzi dvomi telesami. Knižnica ponúka veľké množstvo druhov spojov, no jediný, ktorý je zahrnutý v práci, je *mouseJoint* umožňujúci interakciu užívateľa s aplikáciou prostredníctvom uchopovania objektov.

- **`common.Vec2`** Vektor.

  - **`float x, y`** Súradnice vektoru.

- **`common.Rot`** Matica rotácie, reprezentovaná hodnotami $\sin\alpha$ a $\cos\alpha$.

- **`common.Transform`** Objekt definujúci transformáciu telesa (otočenie a posun). Jedná sa o afinné zobrazenie.

  - **`Vec2 p`** Vektor posunu.
  - **`Rot q`** Matica rotácie.

- **`common.Settings`** Trieda uchovávajúca nastavenie systému.

- **`collision.shapes.Shape`** Abstraktná definícia tvaru.

- **`collision.shapes.CircleShape`** Kruhový tvar. Dedí od triedy *Shape*.

- **collision.shapes.PolygonShape** Polygón. Dedí od triedy *Shape*.
- **collision.shapes.EdgeShape** Hrana. Dedí od triedy *Shape*.
- **dynamics.contacts.Contact** Kontakt medzi dvomi telesami.
  - **Fixture m_fixtureA, m_fixtureB** Kolidujúce predmety.
  - **float m_friction** Trenie medzi kolidujúcimi predmetmi.
  - **float m_restitution** Intenzita odrazenia.

2.2 Inicializácia sveta

Na začiatku je potrebné vytvoriť simulačné prostredie (svet).

```
World w = new World(new Vec2(0, -9.81f));
```

Konštruktor prijíma v parametri vektor gravitačného zrýchlenia. Na vkladanie objektov do simulácie je nutné definovať šablónu:

```
BodyDef bd = new BodyDef();
bd.setType(BodyType.STATIC);
Body ground = w.createBody(bd);
```

Premenná *bd* reprezentuje definíciu telesa, ktorá mu pomocou metódy *setType()* definuje statický typ. Príkaz *createBody()* vytvorí v danom svete nové teleso podľa šablóny.

```
EdgeShape edge = new EdgeShape();
edge.set(new Vec2(-10, 0), new Vec2(10, 0));
Fixture f1 = ground.createFixture(edge, 0.0f);
```

Do telesa je potrebé nadefinovať predmety. Premenná *edge* definuje hranu medzi dvomi bodmi, ktorá je následne vložená ako predmet do telesa *bd*.

```
bd.type = BodyType.DYNAMIC;
bd.position = new Vec2(0.0f, 5.0f);
Body target = w.createBody(bd);
PolygonShape cube = new PolygonShape();
cube.setAsBox(5.0f, 5.0f);
Fixture f2 = newBody.createFixture(cube, 1.0f);
f2.m_friction = 0.2f;
f2.m_restitution = 0.0f;
```

Šablóne *bd* je možné predefinovať hodnoty a použiť ju znovu. Rovnaký proces je použitý na definovanie dynamického telesa *target* v tvare štvorca 10x10 (definované $\frac{a}{2}$) umiestneného do telesa *ground*. Predmetu je naviac definované trenie a odrazivosť.

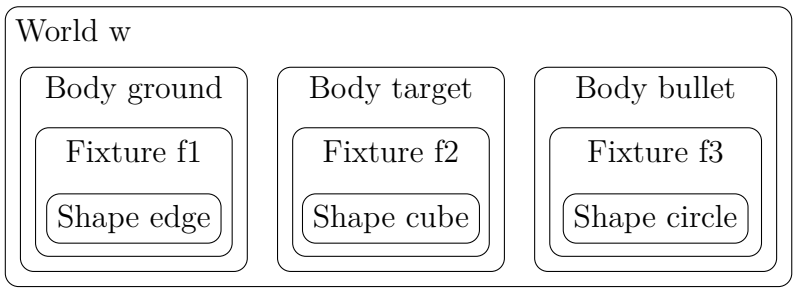
```
bd.type = BodyType.DYNAMIC;
bd.position = new Vec2(-30.0f, 5.3f);
bd.linearVelocity = new Vec2(100.0f, 0.0f);
Body bullet = w.createBody(bd);
CircleShape circle = new CircleShape();
circleShape.m_radius = 1.0f;
Fixture f3 = bullet.createFixture(circle, 2.0f);
f3.m_friction = 0.4f;
f3.m_restitution = 0.1f;
```

Ako posledné teleso je definovaný rýchlo letiaci objekt v tvare kruhu. Tvar je určený polomerom, teleso má nastavenú pozíciu pričom vektor pohybu a predmetu upravuje atribúty trenie a odrazivosť.

Po nadefinovaní sveta spolu s telesami je potrebné dané prostredie simulovať v čase. Na posun simulácie v čase slúži metóda:

```
w.step(m, velocity, iterations);
```

ktorá ju posunie o m milisekúnd dopredu. Parametre *velocity* a *iterations* slúžia na určenie presnosti výpočtu kolízií medzi telesami. V simuláciách sa daný príkaz spolu s vizualizáciou scény bude spúšťať v pravidelných časových intervaloch.



Obr. 2.1: Schéma inicializovaného sveta

Pre získanie prístupu k údajom potrebným k vizualizácii simulácie slúži nasledujúci kód:

```
for (Body b = w.getBodyList(); b != null; b = b.getNext()) {
     for (Fixture f = body.m_fixtureList; f != null; f = f.m_next) {
         Shape shape = f.m_shape;
         switch (shape.m_type) {
             case POLYGON:
                 PolygonShape poly = (PolygonShape) shape;
                 for (int i = 0; i < poly.m_count; ++i) {
                     Vec2 v = b.getWorldPoint(poly.m_vertices[i]);
                     //v - súradnica vrcholu
                 }
                 break;
             case CIRCLE:
                 CircleShape circle = (CircleShape) shape;
                 float r = circle.m_radius;
                 Vec2 v = b.getWorldPoint(circle.m_p);
                 //r - polomer, v - súradnica stredu
                 break;
             case EDGE:
                 EdgeShape edge = (EdgeShape) shape;
                 Vec2 v1 = b.getWorldPoint(edge.m_vertex1);
                 Vec2 v2 = b.getWorldPoint(edge.m_vertex2);
                 //v1 - bod 1, v2 - bod 2
                 break;
         }
     }
}
```

For cyklus iteruje cez všetky telesá, ktoré sú súčasťou sveta a pri každom telese všetky predmety, ktoré sú súčasťou iterovaného telesa. Ku každému predmetu je nutné pristupovať špecificky na základe jeho tvaru. Pomocou funkcie $getWorldPoint()$ je možné získať potrebné koordináty vo svetových súradniciach, ktoré je následne možné spracovať vlastným spôsobom.

Obr. 2.2: Vizualizácia definovanej scény

2.3 Rozšírenie

JBox2D poskytuje široké možnosti od definovania pevných telies s množstvom atribútov cez simuáciu kvapalín [4], elastických telies, svetelných lúčov až po filtrovanie kolízií. To, čo však enginu chýba, je podpora fraktúr telies. Tie môžu

byť v reálnom svete vyvolané mnohými príčinami. Práca sa venuje fraktúram vyvolaných ich vzájomnou kolíziou.

3. Voroného diagram

Primárny cieľ práce je implementácia algoritmu slúžiaceho na trieštenie objektov na fragmenty, ktorá by bola aplikovateľná v engine. Technika, ktorá sa používa na tento účel, sa nazýva *voroného fragmentácia* [10], pri ktorej je na objekt podliehajúci fragmentácii aplikovaný *voroného diagram* definujúci jednotlivé fragmenty. Táto technika poskytuje pomerne jednoduché možnosti pri definícii materiálov založenej na špecifickom rozmiestnení ohnísk jednotlivých fragmentov (viď kapitola 7.2) a podáva dobré výsledky.

3.1 Grafy a metrické priestory

Na začiatok je potrebné definovať niektoré základné pojmy z oblasti teórie grafov.

1 Definícia (Graf)**.** *Graf* je dvojica $G = (V, E)$, kde V je neprázdna množina vrcholov (nazývané tiež uzly) a $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$ je množina dvojprvkových podmnožín vrcholov (neorientovaných hrán). Pokiaľ E tvorí množina usporiadaných dvojíc (u, v), kde $u \neq v$, graf označujeme ako *orientovaný*.

2 Definícia (Podgraf)**.** Nech $G = (V, E)$ je graf. Potom $G' = (V', E')$ je *podgraf* grafu G, pokiaľ platí $V' \subseteq V \wedge E' \subseteq E \cap \binom{V'}{2}$. Pokiaľ zároveň platí rovnosť $E' = E \cap \binom{V'}{2}$, hovoríme o *indukovanom podgrafe*.

3 Definícia (Sled)**.** *Sled* v grafe je taká postupnosť vrcholov, pre ktorú platí, že medzi každými dvomi po sebe idúcimi vrcholmi je hrana. Sled je *uzavretý*, pokiaľ začína aj končí v rovnakom bode.

4 Definícia (Stupeň vrcholu)**.** U neorientovaných grafov je *stupeň vrcholu* definovaný ako: $deg(u) = |\{e \in E \mid u \in e\}|$ (počet hrán, ktoré do daného vrcholu zasahujú).

Nasledujúca časť sa venuje rovinným grafom.

5 Definícia (Oblúk)**.** Oblúk je podmnožina roviny tvaru $o = \{\sigma(x) \mid x \in \langle 0, 1\rangle\}$, kde $\sigma : \langle 0, 1\rangle \to \mathbb{R}^2$ je nejaké prosté spojité zobrazenie intervalu $\langle 0, 1\rangle$ do roviny. Body $\sigma(0)$ a $\sigma(1)$ nazývame *koncové body* oblúku σ.

6 Definícia (Rovinné nakreslenie)**.** Nakreslenie grafu $G = (V, E)$ je zobrazenie κ, ktoré každému vrcholu v priradí bod roviny $\kappa(v)$ a hrane $\{i, j\}$ priradí oblúk s koncovými bodmi $\kappa(i)$ a $\kappa(j)$. Zobrazenie je prosté (rôznym vrcholom odpovedajú rôzne body roviny) a žiadny bod $b(v)$ nie je nekoncovým bodom žiadneho oblúku. Graf spolu s týmto zobrazením nazývame *topologický graf*.

7 Definícia (Rovinný graf)**.** Topologický graf je *rovinný*, ak ľubovoľné dva oblúky odpovedajúce hranám e a f $(e \neq f)$ majú spoločné maximálne koncové body.

8 Definícia (Stena)**.** Nech G je rovinný topologický graf. Množinu $A \subset \mathbb{R}^2 \setminus X$ bodov roviny (kde X je množina všetkých bodov všetkých oblúkov nakreslenia grafu G) nazveme súvislou, ak pre ľubovoľné dva body $x, y \in A$ existuje oblúk $o \subset A$ s koncovými bodmi x, y. Relácia súvislosti rozdelí množinu všetkých bodov roviny, ktoré neležia v žiadnom z oblúkov nakreslenia, na triedy ekvivalencie. Tie nazývame *stenami* topologického rovinného nakreslenia grafu G.

9 Definícia (Metrický priestor)**.** *Metrický priestor* je usporiadaná dvojica (X, d), kde X je neprázdna množina a d je zobrazenie $d : X^2 \to \mathbb{R}$ na usporiadaných dvojiciach prvkov X, nazývané tiež *metrika na* X, pre ktoré sú splnené nasledujúce podmienky:

- $d(x, y) \geq 0$ a $d(x, y) = 0 \iff x = y$
- $d(x, y) = d(y, x)$
- $d(x, y) \leq d(x, z) + d(z, y)$

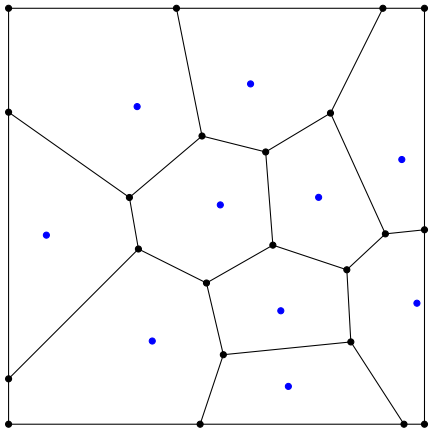
3.2 Voroného diagram

10 Definícia (Voroného diagram)**.** Nech (X, d) je metrický priestor a $V \subset X$ je množina ohnísk $\{v_1, \ldots, v_n\}$. Označme množiny bodov $P = \{P_1, \ldots, P_n\}$, kde $\bigcup P = X$. Potom Voroného diagram je zobrazenie $\tau : (X, V, d) \to P$ ak platí:

$$\forall i \in \mathbb{N}\ \forall x \in P_i\ \forall v \in V : d(x, v_i) \leq d(x, v) \qquad (i \leq n)$$

Túto všeobecnú definíciu zúžime na 2D priestor, za množinu X si zvolíme obdĺžnik a zobrazenie d definujeme ako euklidovskú metriku:

$$d(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$



Obr. 3.1: Voroného diagram

11 Definícia (Konvexita)**.** Nech K je množina bodov v priestore. K je konvexná práve vtedy, keď platí: $\forall x, y \in K : x + k\,(y - x) \in K$ kde $k \in \langle 0, 1 \rangle$.

Pozorovanie. Všetky množiny $P_1, \ldots, P_n$, ktoré sú výslednom voroného diagramu, sú tvorené konvexnými polygónmi.

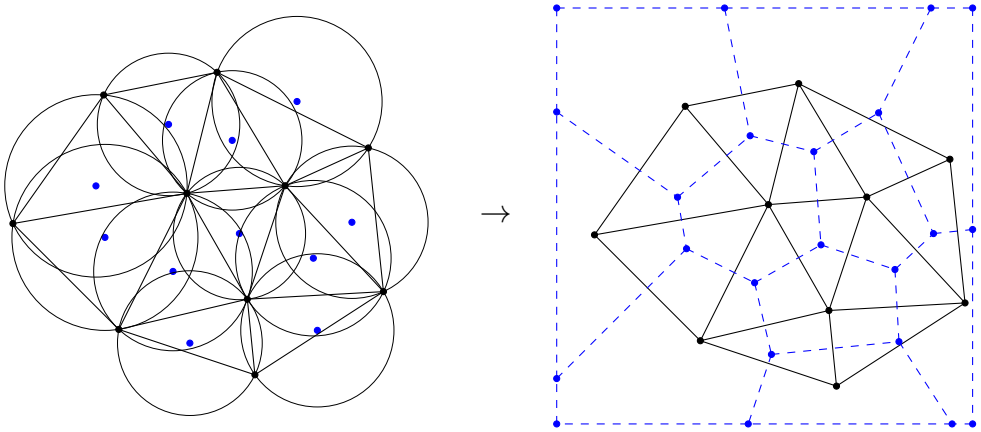
Na voroného diagram je možné nahliadať ako na rovinný graf. Vrcholy fragmentov (konvexných polygónov) tvoria vrcholy grafu, hrany polygónov reprezentujú hrany daného grafu a pod fragmentmi môžeme chápať jednotlivé steny.

12 Definícia (Duálny graf)**.** Nech G je topologický rovinný graf. Označme S ako množinu stien grafu G. Potom graf $G_2 = (V_2, E_2)$ je jeho duál, ak V_2 reprezentuje steny grafu G a $e \in E_2$ práve vtedy, ak medzi stenami existuje hrana z grafu G.

Pozorovanie. Duálny graf k rovinnému grafu je opäť rovinný graf.

13 Definícia (Delaunayho triangulácia)**.** Delaunayho triangulácia je rovinný graf, v ktorom existujú hrany medzi všetkými trojicami vrcholov, pre ktoré platí, že v ich opísanej kružnici sa nenachádza žiadny vrchol.

Pozorovanie. Duálny graf k voroného diagramu je delaunayho triangulácia, ktorá je tiež rovinným grafom [16], pričom vrcholy diagramu sú tvorené stredmi kružníc triangulácie.

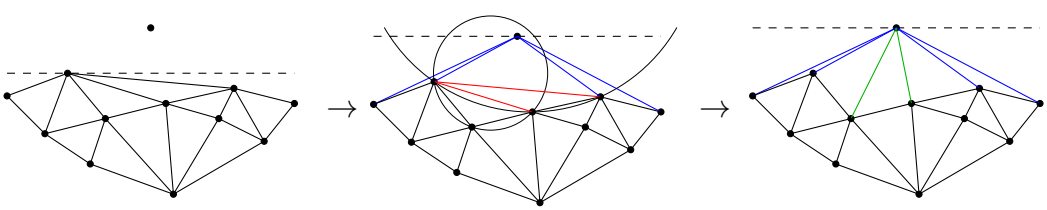


Obr. 3.2: Delaunayho triangulácia a voroného diagram

Pre výpočet voroného diagramu existuje viacero algoritmov, ako napr. aproximačná metóda, fortunov algoritmus [17] alebo algoritmus otáčania hrán. Pre naše účely bol zvolený algoritmus otáčania hrán [12] kvôli jeho jednoduchosti a dobrej amortizovanej časovej zložitosti. Algoritmus slúži na výpočet delaunayho triangulácie [11], ktorá bude následne prevedená v lineárnom čase na jej duál.

3.3 Výpočet delaunayho triangulácie

Vstupom algoritmu je množina bodov (ohnísk) $V \subset \mathbb{R}^2$ a výstupom je množina trojprvkových podmnožín zo vstupnej množiny. Algoritmus pracuje na princípe zametania. Ohniská sa zotriedia v určitom smere (napr. podľa osy y) a následne sa induktívne pridávajú do vypočítanej schémy. Vypočítaná triangulácia bude zároveň ukladať záznam o konvexnom obale. Pri každom kroku platí, že každý vkladaný bod sa nachádza mimo konvexného obalu triangulácie, čo platí aj pre spojnicu vkladaného a posledného vloženého vrcholu. Vkladaný vrchol sa pripojí ku každej hrane obalu tak, aby ho vzniknuté hrany nepretínali. Tým vznikne k nových trojuholníkov, na ktoré sa následne zavolá rekurzívny algoritmus otáčania hrán.



Obr. 3.3: Pridanie vrcholu do delaunayho triangulácie

Algoritmus 1 Delaunayho triangulácia

Vstup: $V[n]$ - pole zotriedených ohnísk
Výstup: $S = \{U_1, \ldots, U_n\}, \forall i : U_i \subseteq V,\ |U_i| = 3$ - triangulácia

1: $S \leftarrow \emptyset$
2: $E \leftarrow \{v_1, v_2\}$ ▷ hrany triangulácie
3: $H \leftarrow \{(v_1, v_2), (v_2, v_1)\}$ ▷ konvexný obal
4: **for all** $v \in V$ **do**
5: **for** $h \in H$ kde $\angle hv < \pi$ **do**
6: $H \leftarrow H \setminus h$
7: $E \leftarrow E \cup \{v, h_1\} \cup \{v, h2\}$
8: flipEdge(h, v) ▷ zaistí delaunayho podmienku
9: **end for**
10: $H \leftarrow H \cup \{(v_l, v_i), (v_i, v_r)\}$ kde v_l a v_r sú hraničné body obalu k v_i
11: **end for**

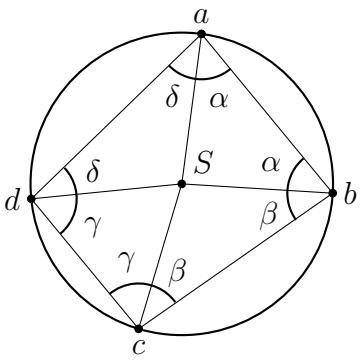
12: **procedure** FlipEdge(Hrana h, Bod v)
13: $T \leftarrow \{h_1, h_2, u\}$ ▷ existujúci trojuholník obsahujúci hranu h
14: **if** v je mimo opísanej kružnice T **then**
15: $S \leftarrow S \cup \{h_1, h_2, v_i\}$
16: **else**
17: $E \leftarrow E \setminus \{h_1, h_2\} \cup \{u, v\}$ ▷ otočí hranu
18: flipEdge$(\{h_1, u\}, v)$
19: flipEdge$(\{h_2, u\}, v)$
20: **end if**
21: **end procedure**

Pre dokázanie korektnosti je nutné popísať trianguláciu štvorice bodov.

Lemma 3.3.1. *Pre každý štvoruholník a, b, c, d ležiaci na kružnici platí, že súčet protiľahlých vnútorných uhlov je π.*

Dôkaz.

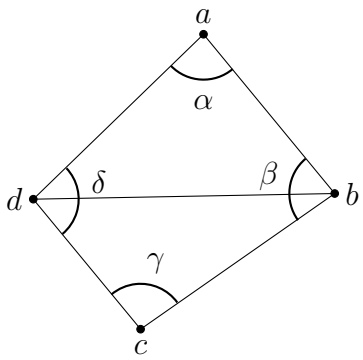
$$2\alpha + 2\beta + 2\gamma + 2\delta = (n-2)\pi \qquad \rightarrow \qquad \alpha + \beta + \gamma + \delta = \pi \qquad (n = 4)$$



Obr. 3.4: Rozdelenie vnútorných uhlov

□

Platí, že ak štvoruholník a, b, c, d má stredovú hranu medzi bodmi b, d, tak delaunayho podmienka je splnená práve vtedy, keď $\alpha + \gamma \leq \pi$ (vyplýva z predošlého lemmatu). Keďže $\alpha + \beta + \gamma + \delta = 2\pi$, otočením hrany dokážeme zaručiť splnenie tejto podmienky aspoň v jednom prípade.



Obr. 3.5: Triangulácia štvoruholníka

Pre trojicu bodov a, b, c je stred opísanej kružnice možné zistiť pomocou vzorca:

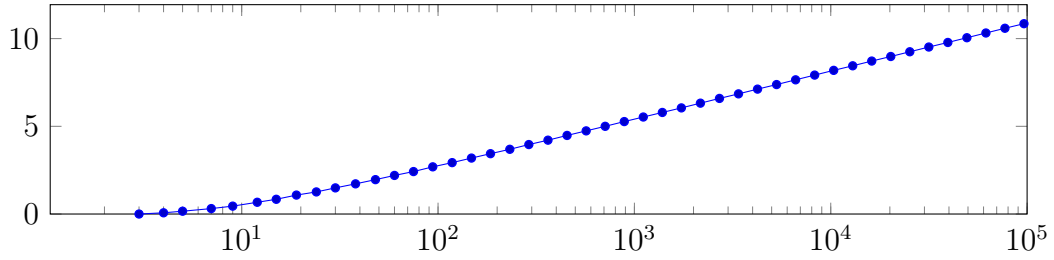
$$
\begin{aligned}
U &= B - A \\
V &= C - A \\
S_x &= \frac{V_y \|U\|^2 - U_y \|V\|^2}{2U \times V} + A_x \\
S_y &= \frac{U_x \|V\|^2 - V_x \|U\|^2}{2U \times V} + A_y \\
r &= \|S - A\| \qquad \text{(euklidovská vzdialenosť bodov)}
\end{aligned}
$$

Norma je definovaná štandardným spôsobom.

Veta 3.3.2. *Počet trojuholníkov delaunayho triangulácie nie je nikdy vyšší, ako 2-násobok počtu vrcholov.*

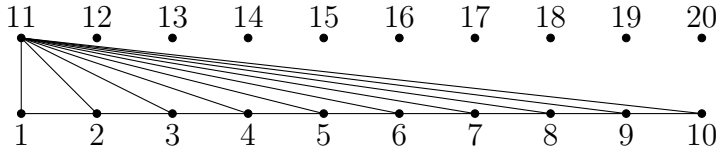
Dôkaz. Indukciou. Najhorší prípad môžeme zostaviť pridávaním vrcholov do trojuholníka triangulácie, ktorý bude následne rozdelený na 3 menšie, čím sa ich počet zväčší o 2. Algoritmus otáčania hrán, ktorý bude volaný na novovzniknutých trojuholníkoch, ich celkový počet neovplyvňuje. □

Pre definovanie časovej zložitosti je nutné zistiť závislosť priemerného počtu otočení hrán na počet vstupných vrcholov. Definovať to matematicky je veľmi náročné, preto nám na túto úlohu poslúži nasledujúci graf:



Obr. 3.6: Závislosť počtu otočení hrán a počtu vstupných vrcholov

Funkčné hodnoty vyjadrujú počet otočení hrán na 1 vrchol ako aritmetický priemer 100 vzoriek náhodne vygenerovaných k vstupných bodov rovnomerne rozmiestnených na štvorcovej ploche. Z grafu možno pozorovať logaritmickú závislosť, čo dáva v priemernom prípade časovú zložitosť $\mathcal{O}(n \log n)$. Najhorší prípad však môže byť až $\mathcal{O}(n^2)$, pretože dochádza až ku kvadratickému počtu otočení hrán.



Obr. 3.7: Najhorší prípad algoritmu otáčania hrán

V uvedenom prípade pri vložení $\left(\frac{n}{2}+k+1\right)$-teho prvku nastáva $\frac{n}{2}-k-1$ otočení hrán pre $k \in \left\{1,\ldots,\frac{n}{2}-1\right\}$, čo dáva amortizovanú časovú zložitosť:

$$\sum_{k=1}^{\frac{n}{2}-1}\left(\frac{n}{2}-k-1\right)=\left(\frac{n}{2}-1\right)^2-\sum_{k=1}^{\frac{n}{2}-1}k=$$
$$\left(\frac{n^2}{2}-1\right)^2-\frac{\left(\frac{n}{2}-1\right)\frac{n}{2}}{2}=\frac{n^2-3n+2}{2}\rightarrow\mathcal{O}(n^2)$$

Najhorší prípad je však možné ošetriť rotovaním zametacej roviny, čím dosiahneme zmenu poradia vstupných vrcholov a minimalizujeme šancu, že daný prípad nastane. To sa neskôr ukáže aj ako efektívne riešenie problému zaokrúhľovania pri floating-point aritmetike u extrémnych vstupoch a zároveň urýchľuje komparátor vrcholov pri triedení.

3.4 Prevod na voroného diagram

Vstupná triangulácia je reprezentovaná poľom vstupných ohnísk a trojicami indexov. Pseudokód algoritmu prevodu bude vyzerať nasledovne:

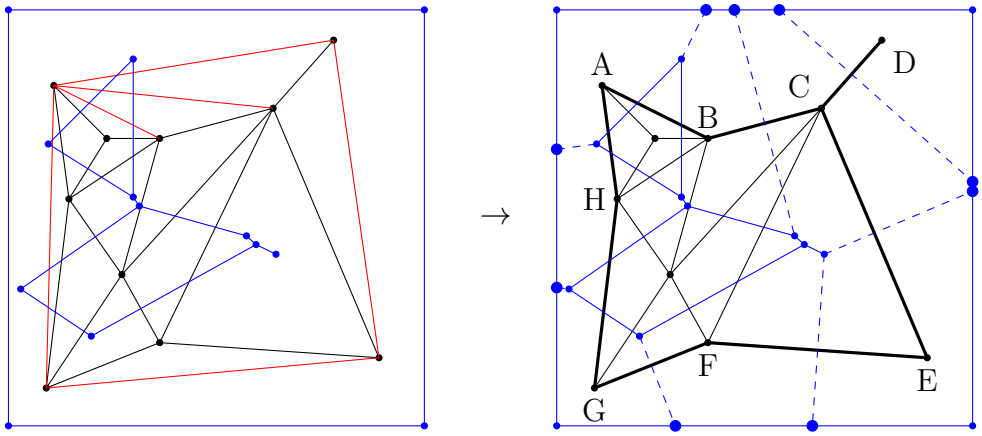
Algoritmus 2 Voroného diagram

Vstup: V, S, E, H - množiny z delaunayho triangulácie; δ - zobrazenie, kde $\delta(T)$ je stred opísanej kružnice T; X - hranica diagramu
Výstup: $[P_1, \ldots, P_n]$ - voroného diagram, V_i je ohnisko P_i

1: $\forall i: \ P_i \leftarrow \emptyset$
2: **for all** $t \in S$ **do**
3: **for** $i, V_i \in t$ **do**
4: P_i.push($\delta(T)$)
5: **end for**
6: **end for**
7: **for all** $h \in H$ **do**
8: addEdge($h, null$) ▷ pridá hraničné body
9: **end for**
10: addCorners(X) ▷ vloží rohy do diagramu
11: **for all** $p \in P$ **do**
12: sort(p) ▷ zotriedi vrcholy polygónov
13: **end for**

14: **procedure** ADDEDGE(Hrana h, Trojuholník t)
15: $t' \leftarrow \{h_1, h_2, v\}$ ▷ trojuholník obsahujúci hranu H, kde $t \neq t'$
16: **if** $\delta(t)$ je vo voroného diagrame (X) **then**
17: $q \leftarrow$ prienik stredovej kolmej polpriamky hrany h s hranicou X
18: P_i.push(q)
19: **else**
20: addEdge($(h_1, v), t'$)
21: addEdge($(v, h_2), t'$)
22: **end if**
23: **end procedure**

V úvode je inicializovaná množinu výstupných polygónov, pričom každý polygón prislúcha práve jednému vstupnému ohnisku. Následne cyklus iteruje cez všetky trojuholníky a do polygónu pre každý bod trojuholníka vloží stred ich opísanej kružnice v prípade, ak daný stred leží v hranici diagramu X. Do diagramu je však nutné pridať vrcholy ležiace na jeho hranici. Odstránením všetkých trojuholníkov triangulácie, ktorých stred opísanej kružnice sa nachádza mimo X získame podgraf, ktorého *ohraničenie* (množina hrán, ktoré sú súčasťou iba jedného trojuholníka) nám určí hľadané body. Ohraničenie tvorí uzavrený sled, ktorý je možné získať rekurzívnou funkciou *addEdge()* podobne ako pri otáčaní hrán. Hľadané body budú tvoriť prienik hranice diagramu a polpriamky kolmej na hranu sledu vychádzajúcej z jej stredu.



Obr. 3.8: Nájdenie hraničných bodov

V uvedenom diagrame je hľadaný sled postupnosť vrcholov: A, B, C, D, C, E, F, G, H, A. V závere stačí priradiť rohové body tým fragmentom, ktoré majú voči nim najbližšie ohnisko a pre každý konvexný polygón usporiadať body podľa uhlu spojnice vrcholu s ohniskom, čím dostaneme výsledný voroného diagram.

4. Konvexná dekompozícia

V JBox2D engine platí, že všetky tvary, ktoré sú reprezentované objektom typu *Shape*, splňujú definíciu konvexity. Dôvod je ten, že detekcia kolízií medzi konvexnými telesami je podstatne rýchlejšia ako u tvarov nekonvexných. Časová zložitosť detekcie prieniku 2 konvexných útvarov je $\mathcal{O}(n+m)$, kde n a m sú počty vrcholov jednotlivých konvexných polygónov. V prípade, že chceme nadefinovať nekonvexné teleso, je nutné ho definovať pomocou konvexných útvarov, ktoré budú pomocou triedy *Fixture* pridané do objektu typu *Body*.

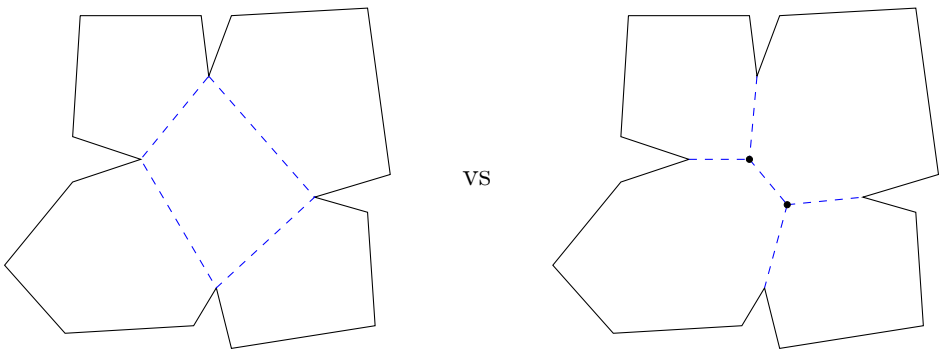
Pri trieštení objektov je prirodzené, že fragmentácii nepodlieha celé teleso, ale len jeho časť v okruhu kolízneho bodu. Pri takomto trieštení môžu vznikať fragmenty, ktoré nespĺňajú definíciu konvexity a preto je potrebné nadefinovať algoritmus, ktorý dokáže nekonvexný polygón rozložiť na skupiny konvexných polygónov, z ktorých by bolo následne pomocou vyššie uvedených nástrojov vyskladané cieľové teleso.

Na tento účel slúži tzv. *konvexná dekompozícia*. Cieľom je rozdeliť nekonvexný polygón na čo najmenšiu množinu disjunktných konvexných útvarov. Snaha o minimalizáciu počtu výsledných konvexných útvarov vedie k zvýšeniu efektivity pri detekcii a spracovaní kolízií počas simulácie.

Uvedené informácie o tejto problematike sú čerpané z knihy *Computational Geometry in C* [18].

4.1 Optimálna dekompozícia

Na riešenie úlohy dekompozície sú k dispozícii 2 možné prístupy - s pridávaním a bez pridávania vrcholov:



Obr. 4.1: Konvexná dekompozícia bez a s pridávaním vrcholov

Nájdenie optimálnej konvexnej dekompozície (minimálny počet konvexných partícií) je výpočtovo ďaleko náročnejšie, ako nájdenie neoptimálneho riešenia. Algoritmus na nájdenie optima pri ľubovoľných segmentoch bol vyriešený s časovou zložitosťou $\mathcal{O}(n^3)$ [21]. Je preto efektívnejšie zvoliť neoptimálne riešenie pomocou triangulácie a *Hertel-Mehlhornovho* algoritmu [22], u ktorého sa počet prvkov dekompozície približuje optimu a funguje s časovou zložitosťou $\mathcal{O}(n \log n)$.

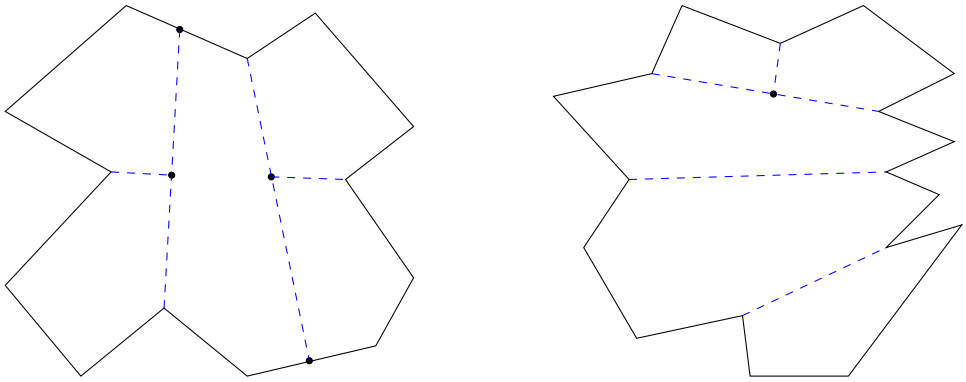
Pre vyhodnocovanie efektivity dekompozície je dôležité vysloviť tvrdenie pre optimálny prípad.

14 Definícia. Vrchol v, ktorý je súčasťou polygónu P, označíme ako *vypuklý*, pokiaľ je jeho vnútorný uhol väčší ako π.

Veta 4.1.1 (Chazelle)**.** *Nech ϕ je najmenší počet konvexných partícií, na ktoré je možné polygón rozdeliť. Pre polygón, ktorý má r vypuklých vrcholov, platí:*

$$\left\lceil \frac{r}{2} \right\rceil + 1 \leq \phi \leq r + 1$$

Dôkaz. V prípade, že každý vypuklý vrchol rozdelí útvar na 2 menšie, dostávame horný odhad počtu partícií $r + 1$. V optimálnom prípade dokážeme spojiť maximálne 2 vypuklé vrcholy, ktoré rozdelia existujúci segment na 2, čo dáva dolný odhad počtu konvexných partícií $\lceil \frac{r}{2} \rceil + 1$ (Chazelle, 1985) [21].



Obr. 4.2: Príklad horného a dolného odhadu optimálnej dekompozície

□

Algoritmus dekompozície, ktorý bude následne popísaný, je zostavený z dvoch hlavných častí:

- Triangulácia polygónu
- Zlúčenie trojuholníkov do konvexných útvarov

4.2 Triangulácia polygónu

Triangulácia je rozdelená na 3 časti:

- Rozdelenie polygónu na monotónne partície
- Triangulácia partícií
- Otáčanie hrán

4.2.1 Rozdelenie polygónu na monotónne partície

15 Definícia (Monotónny reťazec)**.** *Monotónny reťazec* v závislosti na línii L je taká postupnosť hrán, u ktorej platí, že každá priamka rovnobežná s líniou L pretína daný reťazec najviac v jednom bode.

16 Definícia (Monotónny polygón)**.** Polygón v závislosti na línii L je *monotónny*, ak je ho možné rozdeliť na 2 reťazce, ktoré sú k danej línii monotónne.

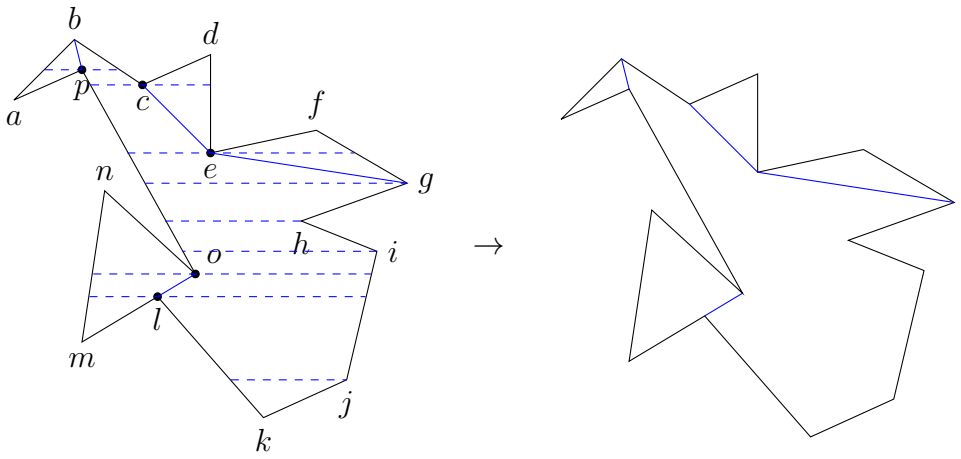
17 Definícia (Hrot)**.** *Hrot* polygónu je vrchol, ktorého susedné vrcholy ležia oba nad alebo pod ním v závislosti na línii L.

Pozorovanie. Polygón je monotónny práve vtedy, keď neobsahuje hrot.

Triangulácia monotónneho polygónu je výrazne jednoduchšia, ako všeobecného. Funguje v čase $\mathcal{O}(n)$ v závislosti na počte jeho vrcholov. Riešením je preto jeho rozdelenie na skupinu monotónnych partícií, ktoré budú triangulované samostatne.

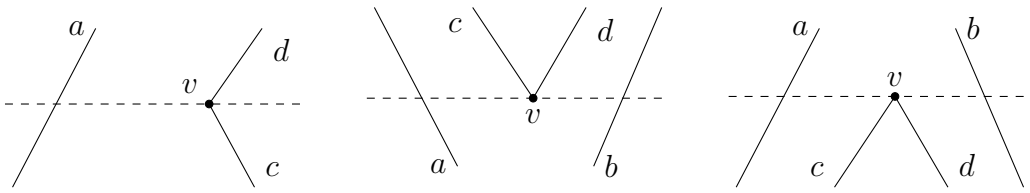
Monotónne polygóny sú definované vzhľadom k nejakému vektoru. Pre jednoduchosť implementácie zvoľme vektor $(1, 0)$. Bude teda platiť, že každá vodorovná priamka pretína monotónny polygón maximálne v 2 bodoch.

Rozdelenie polygónu na monotónne partície rieši tzv. *trapezoidalizácia* [20]. Polygón rozdeľuje na skupinu lichobežníkov, ktoré budú následne zjednotené do monotónnych polygónov. Pre výpočet sa využíva princíp zametacia.



Obr. 4.3: Trapezoidalizácia

Na začiatku sú jednotlivé body zotriedené podľa y-ovej osi a posúvaním zametacej priamky sa pridávajú a odoberajú hrany, ktoré z/do daného vrcholu vedú. Hrany sú ukladané do zoznamu a sú zotriedené podľa x-ovej súradnice ich prieniku so zametacou priamkou (hrany sa neprekrývajú). Pre zachovanie časovej zložitosti $\mathcal{O}(n \log n)$ je možné tento zoznam reprezentovať červeno-čiernym stromom. Pri zametaní môžu nastať 3 typy udalostí:



(a) nahradenie c za d (b) pridanie c a d (c) zmazanie c a d

Obr. 4.4: Udalosti zametacej priamky

Na uvedenej schéme budú hodnoty v zozname pri jednotlivých udalostiach nasledovné:

$$(e_{lk}, e_{kj})$$
$$(e_{lk}, e_{ji})$$
$$(e_{mn}, e_{ml}, e_{lk}, e_{ji})$$
$$(e_{mn}, e_{ji})$$
$$(e_{mn}, e_{no}, e_{po}, e_{ji})$$
$$(e_{mn}, e_{no}, e_{po}, e_{ih})$$
$$(e_{mn}, e_{no}, e_{po}, e_{hg})$$
$$(e_{po}, e_{hg})$$
$$\ldots$$

V priebehu výpočtu bude algoritmus spájať každý hrot s protiľahlým vrcholom, ktorý je súčasťou polygónu. Tým docielime jeho rozklad na monotónne partície.

4.2.2 Triangulácia monotónnych partícií

Podľa vyššie uvedeného, každá partícia je triangulovaná nezávisle. Účelom triangulácie polygónu je pridanie $n - 3$ *diagonál* - hrán spájajúcich jeho nesusedné vrcholy, ktoré sa nepretínajú so žiadnou inou hranou (vrátane samotných diagonál).

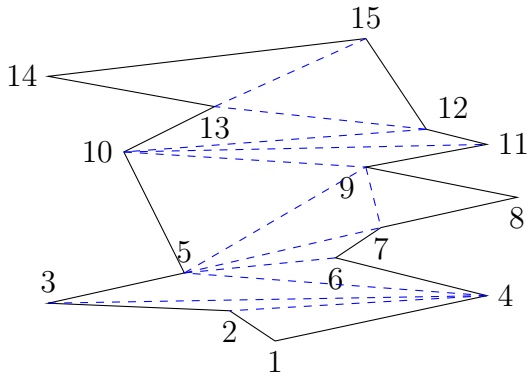
Algoritmus 3 Triangulácia monotónnej partície

Vstup: $V[n]$ - mergované hodnoty pravého a ľavého reťazca; δ - relácia, $\delta(v)$ platí, ak je v z pravého reťazca
Výstup: $D = \{U_1, \ldots, U_n\}, \forall i : U_i = \{u, v\},\ \delta(u) \oplus \delta(v)$

1: $D \leftarrow \emptyset$
2: $Z \leftarrow \langle V_1, V_2 \rangle$ ▷ zásobník
3: **for** $i : 3 \rightarrow n$ **do**
4: **if** $\delta(v_i) \iff \delta(Z.\text{top}())$ **then**
5: **while** $u \leftarrow Z.\text{pop}() \ \wedge\ u$ ide spojiť s v_i **do**
6: $D \leftarrow D \cup \{v_i, u\}$
7: **end while**
8: $Z.\text{push}(u_k)$ kde (u_k, v_i) tvorí poslednú pridanú diagonálu.
9: **else**
10: **while** $u \in Z$ **do**
11: $D \leftarrow D \cup \{v_i, u\}$
12: **end while**
13: $Z \leftarrow \langle Z.\text{top}() \rangle$
14: **end if**
15: $Z.\text{push}(v_i)$
16: **end for**

Na začiatku bude inicializovaný zásobník s prvými dvomi vrcholmi. Algoritmus prechádza vrcholy oboch reťazcov v zotriedenom poradí. Pokiaľ je nový

vrchol z rovnakého reťaza, ako posledný vrchol zo zásobníka, pridáva diagonály s vrcholmi zásobníka od jeho konca, pokiaľ je to možné. Ak z rovnakého reťazca nie je, spojí nový vrchol so všetkými vrcholmi v zásobníku. Na konci kroku budú v zásobníku len neuzavreté vrcholy.



Obr. 4.5: Triangulácia monotónnej partície

Keďže algoritmus preiteruje n-vrcholov a vloží $n-3$ hrán, jeho časová zložitosť je $\mathcal{O}(n)$.

R. Seidel (1991) [20] vo svojej práci ponúka algoritmus s časovou zložitosťou $\mathcal{O}(n \log^* n)$ fungujúci na rovnakom princípe, no naviac zefektívňuje proces trapezoidalizácie.

4.2.3 Otáčanie hrán

Obecný prevod medzi dvomi trianguláciami nekonvexného polygónu za použitia algoritmu otáčania hrán je NP-úplny problém [13]. C. Lawson (1972) [14] však dokazuje, že zaručenie delaunayho podmienky pre ľubovoľnú trianguláciu polygónu je možné realizovať v časovej zložitosti $\mathcal{O}(n + k^2)$, kde k je počet vypuklých vrcholov. To zovšeobecňuje známy fakt, že prevod medzi ľubovoľnými dvomi trianguláciami konvexných polygónov funguje pri najhoršom v čase $2n - 10$ pre $n > 12$ [15].

4.3 Hertel-Mehlhornov algoritmus

Berúc do úvahy, že trojuholník je implicitne konvexný, je v tejto fáze konvexná dekompozícia už k dispozícii. Aplikácia takejto dekompozície by však spôsobovala výrazné spomalenie simulácie kvôli spracovaniu vyššieho počtu vrcholov pri detekcii kolízií medzi telesami. Je preto efektívne zlúčiť trojuholníky do väčších konvexných útvarov. Na to slúži *Hertel-Mehlhornov algoritmus* [22].

18 Definícia. Nech d je hrana oddeľujúca 2 disjunktné konvexné útvary. Pokiaľ odstránenie hrany vytvorí z daných útvarov nekonvexné teleso, hrana d je *základová*. V opačnom prípade je hrana *nezákladová*.

Pozorovanie. Pokiaľ je hrana d základová, platí, že $\exists v \in d : \angle v > \pi$ kde $\angle v$ je braný v rámci daných partícií.

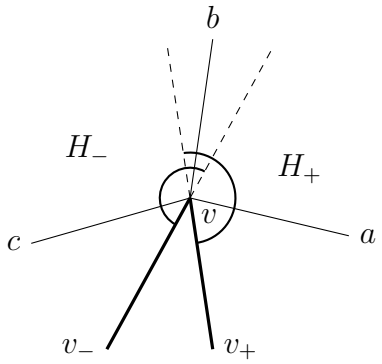
Algoritmus začína na vstupe s triangluláciou P a odstraňuje nezákladové diagonály, pokiaľ existujú. Platí tu však nejednoznačnosť. Výstup aj počet výsledných konvexných útvarov závisí od poradia, v akom budú diagonály odoberané z triangulácie.

Obr. 4.6: Nejednoznačnosť Hertel-Mehlhornovho algoritmu

Pri použití vhodných dátových štruktúr je možné dosiahnuť časovú zložitosť $\mathcal{O}(n)$, takže jediná otázka zostáva v určení efektivity oproti optimálnemu riešeniu. Je preto potrebné vysloviť nasledujúce tvrdenie:

Lemma 4.3.1. *Do každého vypuklého vrcholu vstupujú najviac 2 základové hrany.*

Dôkaz. Nech v je vypuklý vrchol a v_+ a v_- ich priľahlé vrcholy. V polrovine H_+ od hrany vv_+ môže existovať najviac jedna základová diagonála. V prípade, ak by boli 2, jedna z nich by mohla byť odstránená bez porušenia konvexity na vrchole v. To isté platí aj pre opačnú polrovinu H_- od hrany vv_-. Polroviny H_+ a H_- pokrývajú celý vnútorný uhol vrcholu v, z čoho plynie, že z vrcholu nemôžu vychádzať viac ako 2 základové diagonály.



Obr. 4.7: Polroviny vypuklého vrcholu

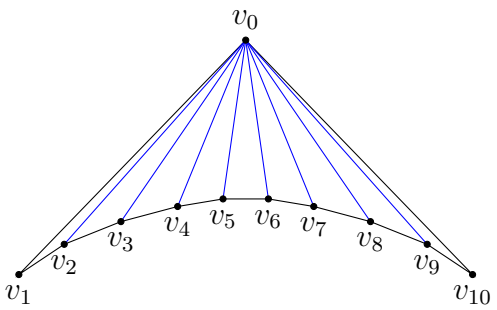
□

Veta 4.3.2. *Hertel-Mehlhornov algoritmus nie je nikdy horší, ako 4-násobok optimálneho odhadu počtu konvexných útvarov.*

Dôkaz. Po ukončení algoritmu platí, že každá existujúca diagonála je základová (pri odstránení ktorejkoľvek z nich sa poruší konvexita). Podľa predošlého lemmatu môže byť každý vypuklý vrchol súčasťou maximálne dvoch základových diagonál, z čoho plynie, že počet nezákladových diagonál nemôže byť väčší ako $2r$, kde r je počet vypuklých vrcholov. Definujme si počet konvexných útvarov výstupu Hertel-Hehlhornovho algoritmu ako ψ. Bude platiť $\psi \leq 2r + 1$. Podľa vety 4.1.1 platí $\lceil \frac{r}{2} \rceil + 1 \leq \phi$, teda $\psi \leq 2r + 1 < 2r + 4 \leq 4\phi$. □

4.3.1 Najhorší prípad

Prípad, kedy je riešenie neúčinné, nastáva, ak existuje postupnosť $n-3$ vypuklých vrcholov. Takýto prípad však nie je možné efektívnejšie vyriešiť ani za použitia optimálnej dekompozície, čo môže spôsobiť zníženie výkonnosti simulácie.



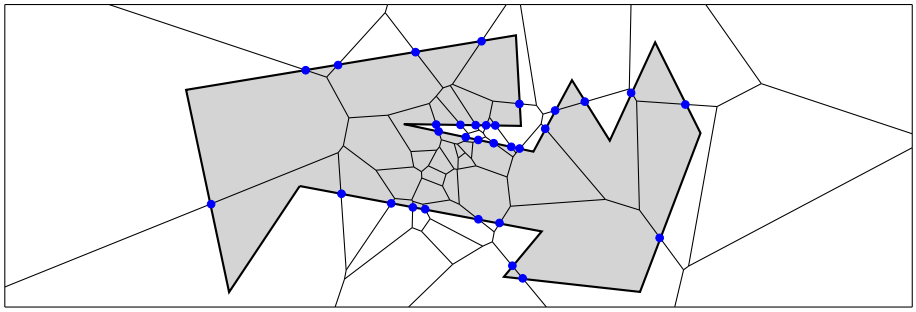
Obr. 4.8: Najhorší prípad konvexnej dekompozície

5. Fragmentácia

Táto kapitola sa zaoberá aplikovaním voroného diagramu na polygóny v spojitosti s kolíziou telies. Pri riešení problému fragmentácie je potrebné vypočítať prienik nekonvexného polygónu s voroného diagramom definujúcim hranice jednotlivých fragmentov, na ktoré sa teleso rozdelí. V drvivej väčšine prípadov však pri fraktúrach telies nedochádza k fragmentácii celého telesa, ale len jeho malej časti, z ktorej sa odštiepia úlomky. Na určenie úlomkov budú aplikované príslušné filtre. Ostatné fragmenty voroného diagramu budú tvoriť pôvodné teleso a podstúpia zjednotenie.

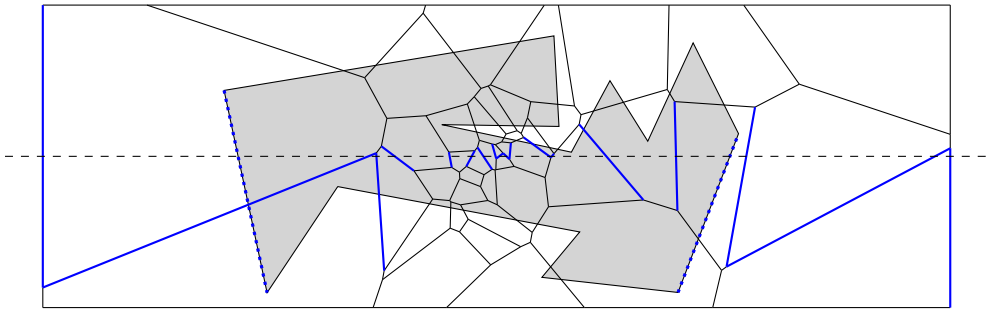
5.1 Prienik polygónu a voroného diagramu

Na riešenie tejto úlohy je v úvode potrebné nájsť všetky body prieniku diagramu s polygónom a vložiť ich do pôvodných útvarov pre ich ďalšie spracovanie. Riešenie bude vychádzať z predpokladu, že sa primárny polygón nachádza vo vnútri voroného diagramu. Je totiž potrebné, aby fragmentácia pokrývala celé teleso. Preto sa hranice diagramu určia podľa rozmerov telesa tak, aby bol predpoklad splnený.



Obr. 5.1: Prienik voroného diagramu a polygónu

Jedno z možných riešení je kontrola prienikov všetkých kombinácií dvojíc hrán z polygónu a diagramu. Toto riešenie je však značne neefektívne, čo nás vedie k použitiu princípu zametania. Pri inicializácii sa vytvoria 2 zoznamy, z ktorých jeden bude ukladať hrany polygónu a druhý hrany diagramu, ktoré sú v prieniku so zametacou rovinou. Pri vložení hrany sa vždy overí existencia prieniku so všetkými hranami nachádzajúcimi sa v druhom zozname.



Obr. 5.2: Hľadanie bodov prieniku pomocou zametacej priamky

Definujme úsečky $|AB|$ a $|CD|$. Prienik je možné zistiť nasledovne:

$$
\begin{aligned}
U &= B - A \\
V &= D - C \\
d &= U \times V && (d = 0 \rightarrow \text{úsečky sú rovnobežné}) \\
t_u &= \frac{C \times V - A \times V}{d} && (t_u \in \langle 0, 1\rangle \rightarrow \text{prienik priamok je na úsečke } |AB|) \\
t_v &= \frac{C \times U - A \times U}{d} && (t_v \in \langle 0, 1\rangle \rightarrow \text{prienik priamok je na úsečke } |CD|) \\
I &= A + Ut_1 = C + Vt_2 && (\text{Bod prieniku priamok})
\end{aligned}
$$

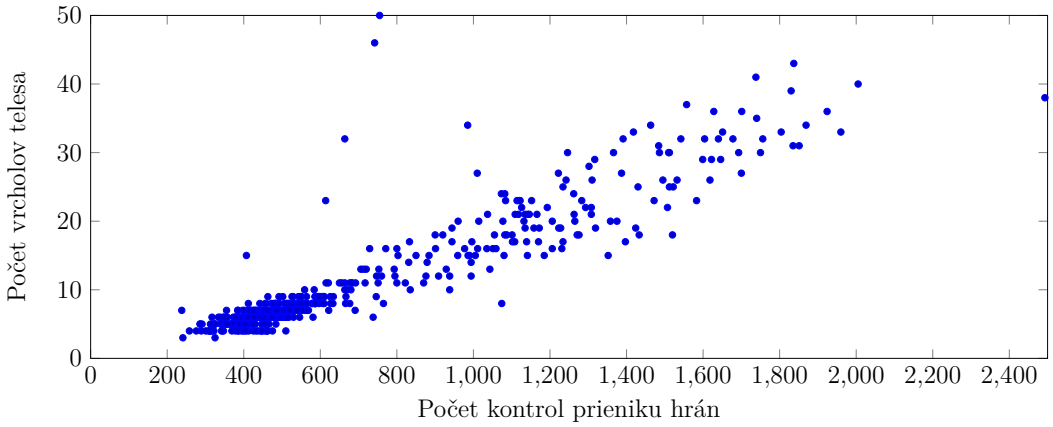
Zápis algoritmu v pseudokóde vyzerá nasledovne:

Algoritmus 4 Prienik polygónu s voroného diagramom

Vstup: H_p - hrany polygónu; H_d - hrany voroného diagramu

```
1:  Z_p, Z_d ← ∅
2:  S = H_p^1 ∪ H_p^2 ∪ H_d^1 ∪ H_d^2                ▷ udalosti zametacej priamky
3:  sort(S);
4:  for all v ∈ S do
5:      h ← hrana, z ktorej je vrchol v
6:      if v je z primárneho polygónu then
7:          if v je počiatočný vrchol hrany h then
8:              Z_p.push(h)
9:              for all h_d ∈ Z_d do
10:                 if h, h_d sa pretínajú then
11:                     rozdelí hrany h, h_d ich priesečníkom
12:                 end if
13:             end for
14:         else
15:             Z_p.remove(h)
16:         end if
17:     else
18:         analogicky s opačnými zoznamami Z_p, Z_d
19:     end if
20: end for
```

Časová zložitosť najhoršieho prípadu je $\mathcal{O}(nm)$, kde n a m reprezentujú počet hrán polygónu a voroného diagramu. Situácia by nastala, ak by existovala priamka $y = c$, ktorá by pretínala všetky hrany. V takom prípade by musela byť overená existencia prieniku medzi každou dvojicou hrán diagramu a polygónu. V priemernom prípade však počet kontrol prieniku dvoch hrán je výrazne menší, čo v kombinácii s faktom, že počet vrcholov polygónu je počas simulácie v priemere iba niekoľko desiatok, dáva dobré časové výsledky, ako ukazuje nasledujúci graf:



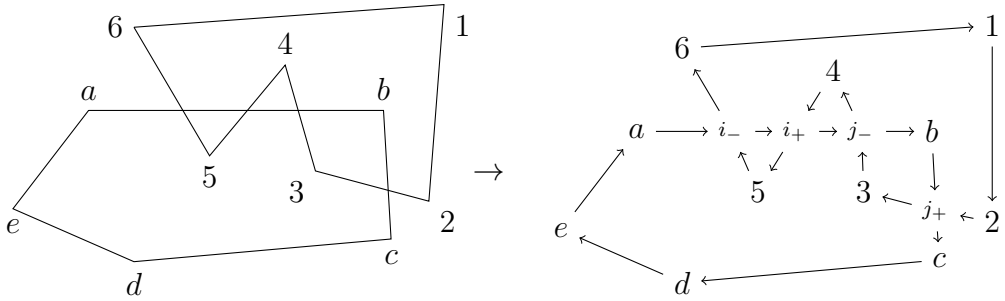
Obr. 5.3: Vzorky fragmentácii z testovacích scenárov

Graf je vyhodnotený z testovacích scenárov pri aplikácii voroného diagramu s fixnou veľkosťou 100 ohnísk (301 hrán diagramu).

Body prieniku hrán budú následne pridané do primárneho polygónu a tiež do jednotlivých konvexných fragmentov voroného diagramu. V tejto fáze sú dáta pripravené na zisťovanie prieniku polygónov. Fragmenty diagramu rozdelíme do 3 skupín podľa toho, či:

1. ležia na jeho hranici,
2. ležia celým obsahom v telese,
3. ležia celým obsahom mimo telesa.

Na začiatku zisťujeme prítomnosť bodov prieniku v jednotlivých fragmentoch. Pokiaľ existujú, je zrejmé, že fragment patrí do 1. skupiny, čo spustí výpočet ich prieniku s hlavným telesom.



Obr. 5.4: Prienik polygónov

Algoritmus začína v prvom bode vstupného prieniku (i_+) a iteruje cez vrcholy fragmentu. Pri iterácii zapisuje jednotlivé vrcholy do listu reprezentujúceho fragment prieniku. Ak počas prechádzania narazí na prienikový bod, zmení sa polygón iterovania. Po narazení na začiatočný bod listu sa daný list vloží do zoznamu a hľadá ďalší bod vstupného prieniku v danom polygóne (j_+), od ktorého opakuje postup. Po prejdení všetkých vrcholov sú v zozname k dispozícii všetky

fragmenty prieniku. Postup iterovania bude podľa uvedenej schémy nasledovný (podčiarknuté vrcholy tvoria fragmenty prieniku):

$$a, i_-, \underline{i_+, 5, i_-}, i_+, j_-, b, \underline{j_+, 3, j_-, b}, j_+, c, d, e, a$$

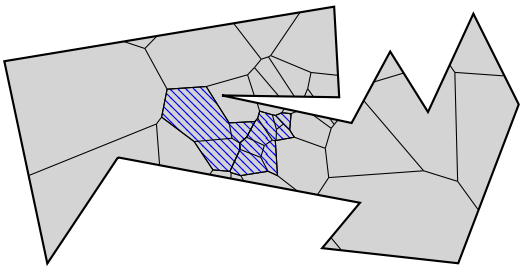
Fragmenty diagramu, ktoré neobsahujú prienikové body, spadajú pod niektorú z ostávajúcich dvoch kategórií. Pre naše účely stačí zistiť, či sa v primárnom polygóne nachádza jeden bod (ohnisko fragmentu). To je možné zistiť vypočítaním počtu prienikov polpriamky vychádzajúcej z daného bodu s polygónom. V prípade, že máme jeden statický polygón s vrcholmi $v^1, \ldots, v^n$ a množinu vyhodnocovaných bodov, je efektívne použiť predspracovanie a pre každú hranu definovať hodnoty m_i, c_i [23]:

$$m_i = \frac{v_x^i - v_x^{i+1}}{v_y^i - v_y^{i+1}}$$
$$c_i = v_x^i - m_i v_y^i$$

Množina bodov nachádzajúcich sa vo vnútri polygónu (x, y) je definovaná ako:

$$\left\{(x,y) \mid \left|\left\{i \mid \left(v_y^i < y \leq v_y^{i+1} \vee v_y^{i+1} < y \leq v_y^i\right) \wedge y m_i + c_i < x\right\}\right| \nmid 2\right\}$$

Zjednotenie fragmentov prieniku a fragmentov 2. kategórie tvoria výsledný hľadaný prienik.



Obr. 5.5: Redukované a vnútorné fragmenty tvoriace výsledný prienik

5.2 Filtrovanie fragmentov

Filtrovanie nastáva dvomi spôsobmi:

- podľa viditeľnosti
- podľa hranice pôsobnosti

5.2.1 Filter viditeľnosti

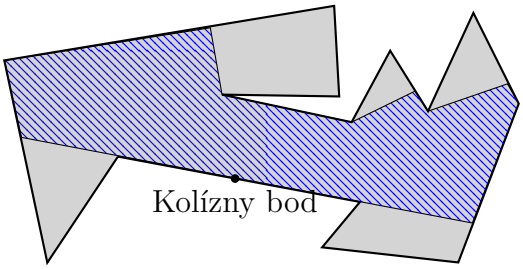
Nech P je primárny polygón a v je bod ležiaci na niektorej z jeho hrán (braný ako kolízny bod). Definujme polygón viditeľnosti $P' \subseteq P$ ako množinu:

$$P' = \{u \mid \forall t \in \langle 0, 1\rangle : \; v + t(u - v) \in P\}$$

Pre efektívnu implementáciu je dobré vysloviť nasledovné lemma:

Veta 5.2.1. *Pokiaľ je polygón P konvexný, platí $P = P'$.*

Dôkaz. Plynie z definície konvexity. □



Obr. 5.6: Polygón viditeľnosti

5.2.2 Filter hranice pôsobnosti

Výber fragmentov je možný aj podľa filtra pôsobnosti. Ten sme si definovali ako prienik polelipsy kombinovaný s opačným polkruhom, ktorý poskytuje realistickejšie výsledky pri testovacích simuláciách (viď kapitola 7.3). Definujme si kolízny bod d, vektor nárazu v a polomer fragmentácie r. Množinu bodov patriacich do nami definovaného útvaru je možné vyjadriť ako:

$$\begin{aligned} M = \{(x,y) \mid {} & (x,y) = T(x',y')\,; && \text{(transformácia)} \\ & (x'^2 + y'^2 < r^2 \wedge y' < 0) \vee && \text{(polkružnica)} \\ & (x'^2 r^{-2} + y'^2 \|v\|^{-2} < 1 \wedge y' \geq 0)\} && \text{(polelipsa)} \end{aligned}$$

Hodnota (x', y') je inverzná afinná transformácia vektoru (x, y), ktorý je otočený o uhol ϕ a posunutý o vektor v. Klasická afinná transformácia má tvar:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}$$

$$\sin\theta = v_x \|v\|^{-2}$$
$$\cos\theta = v_y \|v\|^{-2}$$

kde po vynásobení uvedených regulárnych matíc dostávame transformačnú maticu

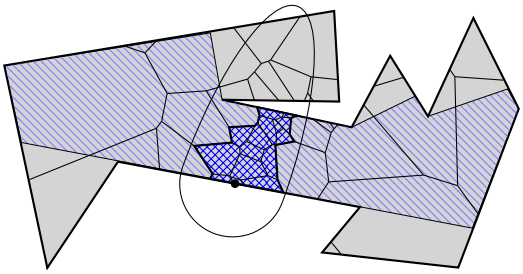
$$T = \begin{pmatrix} \cos\theta & \sin\theta & d_x \\ -\sin\theta & \cos\theta & d_y \\ 0 & 0 & 1 \end{pmatrix}$$

a jej inverz

$$T^{-1} = \begin{pmatrix} \cos\theta & -\sin\theta & -d_x\cos\theta + d_y\sin\theta \\ \sin\theta & \cos\theta & -d_y\cos\theta - d_x\sin\theta \\ 0 & 0 & 1 \end{pmatrix}$$

Vstupný vektor (x, y) transformujeme pomocou matice T^{-1} na vektor (x', y') a pomocou definície množiny M overíme, či sa nachádza v danom útvare. Transformáciu vektorov popisuje *J. Žára* v knihe *Počítačová grafika - princípy a algoritmy* [19]. Definovaním hodnôt r, d a v sa bude zaoberať kapitola 7.1.

Výsledné vyfiltrované fragmenty sú tie, ktoré sa nachádzajú svojim obsahom v oboch filtrovacích útvaroch. Pre dosiahnutie prirodzeného správania simulácie počas štiepenia je tiež potrebné zaručiť súvislosť množiny fragmentov. Tomuto problému sa budeme venovať v kapitole 6.6.



Obr. 5.7: Aplikácia filtrov

5.3 Zjednotenie fragmentov pôvodného telesa

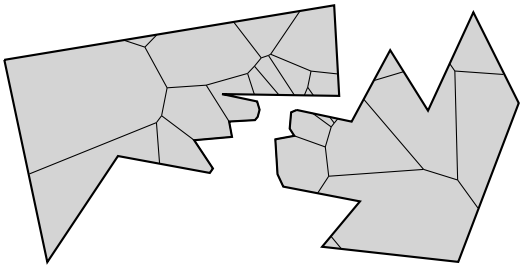
V záverečnej fáze máme k dispozícii 2 množiny fragmentov:

- úlomky
- fragmenty tvoriace pôvodné teleso

Fragmenty tvoriace pôvodné teleso je potrebné zjednotiť. Tie je možné charakterizovať ako rovinný graf $G = (V, E)$, u ktorého je cieľom zistiť podgraf $G = (V', E')$ reprezentujúci jeho ohraničenie. Pre opísanie problému bude potrebné vysloviť definíciu grafovej súvisloti.

19 Definícia (Grafová súvislosť)**.** *Komponent súvislosti* grafu (V, E) je maximálny súvislý podgraf, v ktorom platí, že $\forall v, u \in V$ existuje cesta medzi danými vrcholmi. Graf je *súvislý*, pokiaľ má práve jeden komponent súvislosti.

Definujme zobrazenie $p : V \to N$, ktoré každému vrcholu priradí číslo reprezentujúce počet vnútorných stien grafu, ktorých je daný vrchol súčasťou. Cieľom zjednotenia fragmentov je nájsť taký indukovaný podgraf, ktorý vznikne z grafu G odstránením všetkých vrcholov, pre ktoré platí $p(v) = deg(v) : v \in V$ a hrán s nimi spojenými.



Obr. 5.8: Fragmenty pôvodného telesa

Nájdenie daného ohraničenia môžeme realizovať pomocou výberu hrán. Pre všetky hrany hľadaného podgrafu platí, že sú súčasťou práve jednej steny (fragmentu). To je možné docieliť v čase $\mathcal{O}(m)$, kde m je počet hrán. Zostavením nového grafu získavame výsledné hľadané fragmenty.

Veta 5.3.1. *Ohraničenie fragmentov pôvodného telesa tvorí množina ciest. Ich počet je rovný počtu komponentov súvislosti pôvodného grafu.*

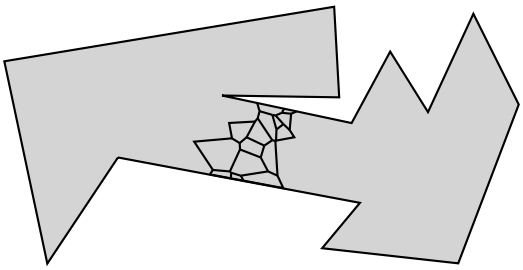
Dôkaz. Prvé tvrdenie dokážeme sporom. Ak by neplatilo, musel by existovať vrchol, z ktorého vedú aspoň 3 hrany. To však nie je možné, pretože by jedna z daných hrán musela byť súčasťou dvoch vnútorných stien, čo z definície nie je možné.

Druhé tvrdenie dokážeme z definície delaunayho triangulácie a grafovej duality, na základe ktorých platí:

$$\forall v \in V : deg(v) \leq 3$$

Preto nemôže existovať vrchol spájajúci 2 cesty, z čoho plynie, že počet ciest je rovný počtu komponentov súvislosti pôvodného grafu. □

Výsledok fragmentácie bude tvorený množinou úlomkov a zjednotenými fragmentami pôvodného telesa.



Obr. 5.9: Výsledok fragmentácie

6. Implementácia

Uvedená kapitola sa zaoberá už konkrétnymi algoritmami pre aplikovanie procesu trieštenia do JBox2D enginu. Všetky algoritmy implementované v práci sú umiestnené v priečinku *org/jbox2d/fracture* a majú uvedenú štruktúru:

- *voronoi* - balík implementujúci voroného diagram
- *hertelmehlhorn* - balík implementujúci Hertel-Mehlhornov algoritmus
- *fragmentation* - balík implementujúci fragmentáciu
- *materials* - balík implementácií materiálov
- *util* - balík s obecnými dátovými štruktúrami
- Základné objekty patriace balíku *org.jbox2d.fracture*

Pre spôsob implementácie je potrebný výber návrhového vzoru.

6.1 Singleton

Singleton je jeden z najjednoduchších a najbežnejších návrhových vzorov využívaných v Jave [24]. Zakladá sa na jednoduchom princípe - po spustení aplikácie sa alokuje jediná inštancia, ktorá bude poskytovať prostriedky pre výpočet. Pri inicializácii sa alokujú všetky potrebné pomocné dátové štruktúry pre daný algoritmus, čím sa redukuje pomalá dynamická alokácia pamäte na haldu a zvyšuje sa tým výkon programu. Pri nízkych pamäťových nárokoch zároveň nie je potrebné riešiť veľkosť alokovanej pamäte, pretože máme záruku, že daný objekt bude v pamäti len jeden krát.

Uvedený návrhový vzor je použitý vo všetkých hlavných algoritmoch, ktoré sme implementovali na výpočet voroného fragmentácie. V prípade snahy o optimalizáciu pomocou paralelného výpočtu na viacerých vláknach procesora je možné tento vzor čiastočne pozmeniť a alokovať pre každé vlákno samostatnú inštanciu. Implementácia voroného fragmentácie však nenesie zvýšené časové nároky v porovnaní s detekciou kolízií počas behu simulácie, a berúc do úvahy, že simulácia pracuje na jednom vlákne, snaha o paralelný výpočet pomocou viacerých vlákien by bola kontraproduktívna. Preto budeme používať v našej implementácii len jednu inštanciu.

6.2 Základné objekty

Pre popis implementácie je nutné uviesť niektoré základné objekty v balíku *org.jbox2d.fracture*:

- `Polygon` Polygón implementovaný listom vrcholov typu *Vec2*.
- `Fragment` Fragment (dedí od triedy *Polygon*), ktorý je výstupom voroného diagramu. Sú nim taktiež reprezentované úlomky telesa, čím ich je možné rozlíšiť od fragmentov pôvodného telesa.

- **`PolygonFixture`** Polygón (dedí od triedy *Polygon*) slúžiaci na vkladanie nekonvexných mnohouholníkov do simulácie. Okrem listu vrcholov uchováva zoznam predmetov (*Fixture*), ktoré vznikli jeho konvexnou dekompozíciou.

- **`Fracture`** Reprezentuje dvojicu telies (primárne a sekundárne), u ktorých dochádza k fragmentácii primárneho telesa. Uchováva všetky potrebné atribúty o kolízii, ktoré sú potrebné pre realizáciu rozpadu primárneho telesa.

- **`Material`** Abstraktná trieda implementujúca materiál. Deklaruje atribúty ako pevnosť, veľkosť úlomkov a ďalšie atribúty potrebných v implementácii. V prípade, že predemetu *Fixture* definujeme atribút *m_material*, teleso bude podliehať trieštenu. Predmetu, ktoré majú tento atribút nastavený na *null*, trieštenu nepodliehajú.

- **`FractureListener`** Rozhranie ponúkajúce možnosť implementovať udalosť, ktorá je volaná vždy po rozpade telesa.

6.3 Implementácia voroného diagramu

Voroného diagram je implementovaný v balíku $org.jbox2d.fracture.voronoi$ 4 triedami:

- **`Hull`** Reprezentuje spojový zoznam konvexného obalu potrebný pre výpočet delaunayho triangulácie.

- **`Edge`** Hrana určujúca spojnicu dvoch bodov a susediacich trojuholníkov delaunayho triangulácie.

- **`Triangle`** Trojuholník delaunayho triangulácie.

- **`SingletonVD`** Hlavný objekt, v ktorom je implementovaný hlavný algoritmus. Obsahuje skupinu premenných, do ktorých sa ukladajú hodnoty z výpočtu po zavolaní procedúr spúšťajúcich výpočet:

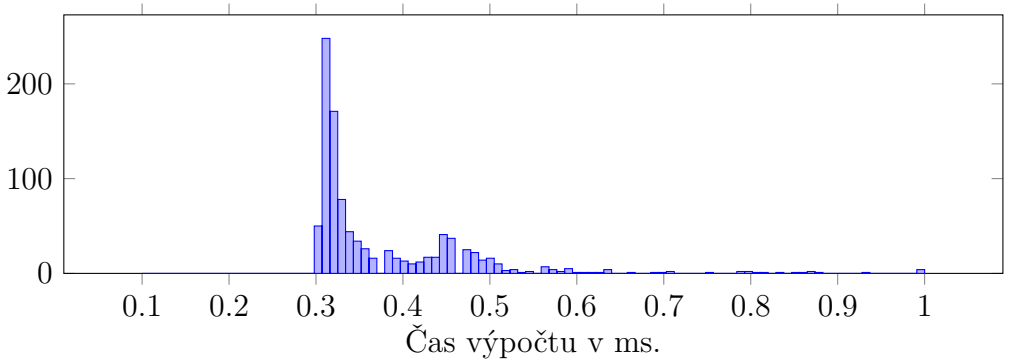
  - **`Vec2[] points`** Pole výstupných bodov voroného diagramu.
  - **`int[][] voronoi`** Výsledný voroného diagram. Prvý index určuje polygón a druhý určuje index vrcholu z poľa *points*. Počet polygónov je rovný počtu vstupných vrcholov, pričom index je zhodný s indexom jeho ohniska zo vstupného poľa, ktorý mu prináleží.
  - **`int[] vCount`** Počet vrcholov jednotlivých polygónov vo voroného diagrame.
  - **`void calculateVoronoi()`** Metóda spúšťajúca primárny výpočet. Obsahuje parametre:
    - **`Vec2[] focee`** Udáva vstupné ohniská.
    - **`Vec2 a, b`** Definujú obdĺžnik, na ktorý bude diagram aplikovaný.

Pre efektívnu implementáciu je výhodné vrcholy reprezentovať indexmi ich umiestnenia vo vstupnom poli. Hlavná motivácia tohto návrhu je efektívny prevod delaunayho triangulácie na duálny graf. Kvôli optimalizácii sú hodnoty určené iba na čítanie a v prípade potreby modifikovať výstupné údaje, je vhodné vytvoriť ich lokálne kópie.

6.3.1 Optimalizácia

JBox2D engine má rovnako, ako aj iné algoritmy, svoje výkonnostné limity. Trieštenie objektu na príliš veľké množstvo fragmentov by spôsobilo spomalenie celého výpočtu. Môžeme preto predpokladať, že počet vstupných vrcholov pre výpočet diagramu nebude väčší ako nejaké fixné n (pre naše účely si definujeme $n = 2^8$). Pri implementácii algoritmu narazíme na úlohu definovania dátových štruktúr ukladajúcich hrany delaunayho triangulácie potrebných pre výpočet. Pre tie je vhodné použiť hašovaciu tabuľku s amortizovanou časovou zložitosťou $\mathcal{O}(c)$ vo všetkých potrebných operáciách a s lineárnou pamäťou. Berúc do úvahy použitie návrhového vzoru *singleton*, je výhodnejšie inicializovať pole o kvadratickej veľkosti $n^2 \cdot 4B = 256kB$, čím sa optimalizuje vyhľadávanie hrán vďaka dokonalému hašovaniu.

Po inicializácii ostatných inštancií potrebných pre výpočet bude veľkosť alokovanej pamäte cca 1MB, čo celkové pamäťové nároky knižnice výrazne neovplyvňí.

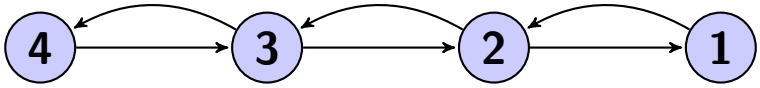


Obr. 6.1: Histogram časov výpočtu voroného diagramu

Uvedené testy sú vykonávané na zariadení s procesorom *Intel(R) Core(TM) i3-2330M* s taktovacou frekvenciou *2.20GHz* a operačným systémom *Windows 7*. Hodnoty reprezentujú časy strávené na výpočet voroného diagramu vygenerovaného z 256 bodov.

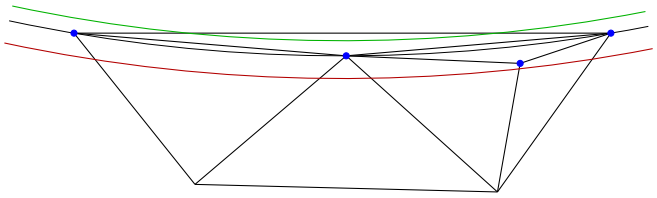
6.3.2 Zaokrúhľovanie a extrémne vstupy

Pri implementácii algoritmov využívajúcich floating-point aritmetiku narazíme na problémy spojené s nepravdepodobnými vstupmi, ktoré dokážu vyvolať nekorektné výsledky. Jedným príkladom je definovanie ohnísk v jednej línii. Pre ošetrenie tohto prípadu treba v algoritme na výpočet delaunayho triangulácie definovať výpočet pre prvých k-vrcholov a vytvoriť dvojitý konvexný obal bez trojuholníkov. Ten zaručí, že po vložení ďalšieho vrcholu definovaný algoritmus vytvorí korektnú trianguláciu.



Obr. 6.2: Konvexný obal línie

Ďalší veľmi špecifický problém môže vzniknúť, ak sa tvar trojuholníka približuje k línii, čo spôsobí, že polomer opísanej kružnice je príliš veľký. To môže pri výpočte spôsobiť odchýlku, kvôli ktorej polomer kružnice bude väčší ako reálny a pri triangulácii sa budú otáčať nesprávne hrany. Pre riešenie tohto problému je efektívne vynásobiť polomer číslom $1 - \epsilon$.



Obr. 6.3: Zaokrúhľovacia odchýlka

To však ešte nezaručuje korektnosť výsledného diagramu pre všetky vstupné hodnoty. Je preto okrem toho dôležité otočiť zametaciu rovinu o náhodný uhol, čím dosiahneme zmenu počiatočného triedenia vstupných vrcholov v algoritme zametania. Transformácia sa dá docieliť vygenerovaním náhodného uhlu v inicializácii singletonu a vrcholy budú triedené podľa ich transformácie maticou:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Okrem ošetrenia extrémnych prípadov minimalizuje pravdepodobnosť výskytu vstupu s kvadratickou časovou zložitosťou (viď kapitola 3.3). Zároveň optimalizuje komparátor, ktorý môže vrcholy triediť len podľa jednej súradnice. Komparátor definujeme ako:

$$\tau = -\sin\theta x + \cos\theta y$$

Uvedený princíp je možné použiť pre všetky algoritmy využívajúce metódu zametania.

6.4 Triangulácia pomocou knižnice poly2tri

Pre implementáciu triangulácie polygónov je v práci použitá knižnica *poly2tri*, ktorá je voľne dostupná na stránkach *GitHub*-u [3] pod licenciou *new BSD* [8]. Poskytuje nástroj na delaunayho trianguláciu polygónov s dierami. Je značne optimalizovaná a má jednoduché rozhranie:

- **`Triangulation`** Objekt realizujúci trianguláciu polygónov. Má jednu kľúčovú metódu:
  - **`ArrayList<ArrayList<Integer>>> triangulate()`**

    so vstupnými parametrami:

* **`int numContures`** Počet vrcholov polygónov: $n \in \mathbb{N}$.
* **`int[] numVerticesInContures`** Počet vrcholov jednotlivých polygónov (platných hodnôt je len prvých n) reprezentovaný n-ticou: $C = (k_1, k_2, \ldots, k_n)$.
* **`double[][] vertices`** Vrcholy jednotlivých polygónov:

$$\begin{aligned} V = \Big(& (x_{11}, y_{11}), (x_{12}, y_{12}), \ldots, (x_{1k_1}, y_{1k_1}), \\ & (x_{21}, y_{21}), (x_{22}, y_{22}), \ldots, (x_{2k_2}, y_{2k_2}), \\ & \ldots \\ & (x_{n1}, y_{n1}), (x_{n2}, y_{n2}), \ldots, (x_{nk_n}, y_{nk_n}) \Big) \end{aligned}$$

Musí platiť, že poradie vrcholov v rámci jednotlivých polygónov bude v protismere hodinových ručičiek.

V uvedenej práci podpora dier nie je, preto majú parametre pri volaní metódy uvedený tvar:

$$\begin{aligned} & n = 1 \\ & C = (k) \\ & V = \Big((x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k) \Big) \end{aligned}$$

Ak je poradie bodov nekorektné, alebo hrany polygónov sa pretínajú, metóda vyhodí výnimku.

Výstup metódy reprezentuje list listov indexov jednotlivých vrcholov zo vstupného parametra *vertices*.

6.5 Implementácia Hertel-Mehlhornovho algoritmu

Algoritmus je implementovaný v balíku $org.jbox2d.fracture.hertelmehlhorn$. Implementácia obsahuje 4 triedy:

- **`TNode`** Reprezentuje spojový zoznam vrcholov konvexných telies. Pre každú dvojicu vrchol-mnohouholník existuje jedna inštancia tohto objektu.
- **`Diagonal`** Diagonála. Je reprezentovaná záznamami pre oba uzly typu *TNode* oboch susediacich mnohouholníkov, ktoré daná hrana oddeľuje. Zároveň uchováva odkazy (indexy) konvexných útvarov, medzi ktorými sa nachádza.
- **`EdgeTable`** Hašovacia tabuľka hrán. Umožňuje vyhľadávať hrany na základe indexov príslušných vrcholov v čase $\mathcal{O}(c)$.
- **`SingletonHM`** Primárny objekt zostavujúci konvexnú dekompozíciu zo vstupnej triangulácie, ktorý obsahuje jednu metódu a jednu výstupnú premennú:
  - **`void calculate()`** Metóda implementujúca Hertel-Mehlhornov algoritmus. Má parametre:

    - `int[][] list` Triangulácia z knižnice *poly2tri*. Je reprezentovaná množinou trojíc indexov vrcholov z poľa *vertices*.
    - `Vec2[] vertices` Vrcholy triangulácie.
    - `int maxVCount` Kvôli pamäťovej alokácii má JBox2D v premennej *maxPolygonVertices* objektu *org.jbox2d.common.Settings* nastavené obmedzenie, ktoré limituje počet vrcholov každého polygónu. Preto bolo nutné algoritmus mierne upraviť a zaručiť, aby počet vrcholov nepresiahol limit.
  - `Polygon[] polygons` Množina polygónov ako výsledok konvexnej dekompozície.

Algoritmus na začiatku inicializuje diagonály, ktoré umiestni do hašovacej tabuľky *EdgeTable*, pričom každý trojuholník vloží do poľa vo formáte spojového zoznamu typu *TNode*. Na poradí trojuholníkov v poli nezáleží. Výpočet sa začne realizovať spustením rekurzívnej metódy, ktorá postupne prechádza každú hranu každého trojuholníka z poľa (pole sa priebežne modifikuje a trojuholníky sa odstraňujú) a zlučuje susedné útvary kým je to možné. Na konci algoritmu zostanú v poli výsledné konvexné útvary. Odstraňujú sa len tie trojuholníky, ktoré ešte neboli preiterované. Algoritmus prejde každý trojuholník práve raz a pri každom vykoná konštantný počet operácii, čo dáva časovú zložitosť $\mathcal{O}(n)$. Počas rekurzie sa zároveň kontroluje limit $maxVCount$.

6.6 Implementácia fragmentácie

Algoritmus je implementovaný v balíku $org.jbox2d.fracture.fragmentation$, ktorý obsahuje 9 tried:

- `AEdge` Abstraktná trieda reprezentujúca hranu polygónu.
- `EdgeDiagram` Hrana voroného diagramu (dedí od *AEdge*).
- `EdgePolygon` Hrana polygónu (dedí od *AEdge*).
- `Arithmetic` Trieda implementujúca aritmetické funkcie pre geometrické výpočty.
- `EVec2` Vrchol hrany slúžiaci pri výpočte prieniku primárneho polygónu a voroného diagramu. Reprezentuje udalosť zametacej priamky.
- `GraphVertex` Objekt využívaný pri zjednocovaní fragmentov pôvodného telesa. Reprezentuje prvok spojového zoznamu výstupných ciest.
- `IContains` Interface, ktorý implementuje filter hranice pôsobnosti. Implementácia sa nachádza v triede *Material*.
- `Vec2Intersect` Prienik hrán primárneho polygónu a voroného diagramu. Je potrebný pre iterovanie pri detekcii fragmentov.
- `Singleton` Trieda, v ktorej prebieha primárny výpočet. Obsahuje jednu kľúčovú metódu:

- **`void calculate()`** Metóda implementujúca fragmentáciu. Má parametre:
  - **`Polygon p`** Primárny polygón.
  - **`Vec2[] focee`** Ohniská voroného diagramu.
  - **`Vec2 contactPoint`** Dotykový bod.
  - **`IContains ic`** Implementácia filtra hranice pôsobnosti.
- **`Polygon[] polygons`** Pole polygónov ako výsledok fragmentácie.

Na začiatku výpočtu sa inicializujú udalosti zametacej roviny objektu *EVec2* pre každý vrchol každej hrany. Pomocou implementácie dvomi zásobníkmi (viď kapitola 5.1) sa detekujú prieniky všetkých hrán objektom *Vec2Intersect* a príslušné vrcholy sa vkladajú do primárneho polygónu a polygónov diagramu. Následne prebehne rozdelenie prienikových polygónov.

V druhom kroku sú prehľadávaním do šírky po polygónoch začínajúc v dotykovom bode filtrované fragmenty do dvoch skupín. Prehľadávanie do šírky zaručuje súvislosť množiny úlomkov, čím docielime prirodzené správanie simulácie.

V závere preiteruje cez všetky hrany grafu zjednocovania a pomocou objektu *GraphVertex* zostaví zvyšok pôvodného telesa, ktoré následne prevedie na typ *Polygon*.

6.7 Proces fragmentácie

Zakomponovanie fragmentácie do enginu si vyžadovalo menšie zmeny v samotnom kóde JBox2D. Simulácia sa posúva v diskrétnych časových jednotkách pomocou metódy *w.step()* (viď kapitola 2.2). Počas každého kroku nastáva detekcia a spracovanie kolízie v objekte. Po jej spracovaní je následne spustená metóda overujúca kritičnosť kolízie implementovaná v objekte *Fracture*. Pokiaľ intenzita (*normalImpulse*) presiahne určitú kritickú hodnotu, algoritmus vyhodnotí kolíziu za fraktúru a vloží ju do hašovacej tabuľky. Kolízia musí spĺňať 2 podmienky - intenzita kolízie musí presiahnuť limit definovaný materiálom (pevnosť) a zároveň obsah telesa musí byť väčší ako nejaká fixná hodnota, ktorá zabráni nekontrolovateľnému rekurzívnemu trieštenu. Druhá podmienka sa vyhodnocuje len v prípade, že sa jedná o dynamické teleso. Ignorovaním tohto faktu by vznikali malé statické čiastočky, ktoré by nebolo možné roztrieštiť. Dôležitá je aj skutočnosť, že modifikácia objektov je povolená až na konci kroku (*step()*) a počas jeho spracovania sú objekty uzamknuté.

Spracovanie fraktúry je zostavené z niekoľkých krokov:

1. Definovanie tvaru trieštiaceho objektu
2. Náhrada trieštiaceho objektu
3. Nastavenie atribútov novým telesám

6.7.1 Definovanie tvaru trieštiaceho objektu

Engine ponúka definíciu niekoľkých tvarov. Pre naše účely budeme používať 2 typy - polygón a kruh. Rozpad polygónu je popísaný v kapitole 5, preto je nutné definovať rozpad kruhu. Kruh je definovaný stredom a polomerom. Jediné

a zároveň najjednoduchšie riešenie je transformácia kruhu na pravidelný polygón s n-vrcholmi, kde n je fixne definované (32). Pre zachovanie obsahov útvarov a dosiahnutie čo najlepšej aproximácie je vhodné definovať nový polomer n-uholníka r', pre ktorý bude platiť $S_k = S_m$, pričom:

$$S_k = \pi r^2 \qquad \text{(obsah kruhu)}$$

$$S_m = r' \sin\frac{\alpha}{2}\, r' \cos\frac{\alpha}{2}\, n, \qquad \alpha = \frac{2\pi}{n} \qquad \text{(obsah mnohouholníka)}$$

Úpravou odvodíme:

$$\pi r^2 = r' \sin\frac{\alpha}{2}\, r' \cos\frac{\alpha}{2}\, n \qquad \rightarrow \qquad \pi r^2 = r'^2 \frac{\sin\alpha}{2} n$$

$$r' = \sqrt{\frac{\pi r^2}{n}\frac{2}{\sin\alpha}} \qquad \rightarrow \qquad r' = r\sqrt{\frac{\alpha}{\sin\alpha}}$$

a súradnice vrcholov budú:

$$v_i = (\sin i\alpha, \cos i\alpha) \qquad \text{pre } 0 \leq i < n, i \in \mathbb{N}$$

Tým prevedieme problém fragmentácie kruhu na problém fragmentácie n-uholníka.

6.7.2 Náhrada trieštiaceho objektu

Pre fragmentáciu použijeme algoritmus popísaný v kapitole 5. Výstupom je množina objektov typov *Polygon* a *Fragment*. Typ určuje, či sa jedná o úlomok alebo zvyšok pôvodného telesa. Následne sa rozpadajúce teleso zmaže, pre každý nový fragment sa vytvorí inštancia typu *Body*, do ktorej sa vložia predmety typu *Fixture* vytvorené na základe konvexnej dekompozície daného fragmentu. Algoritmus dekompozície je popísaný v kapitole 4.

6.7.3 Nastavenie atribútov novým telesám

Po vložení nových telies do simulácie je potrebné týmto telesám (vrátane toho, ktoré vyvolalo fraktúru) nastaviť korektné parametre, ako materiál, typ, pozíciu, otočenie, uhlovú rýchlosť a vektor pohybu.

Premenná *m_material* umožňuje nastaviť telesám úlomkov materiál. V prípade, že je hodnota *null*, úlomky nebudú podliehať ďalšej fragmentácii, čo zabráni rekurzívnemu trieštenu. To sa však netýka fragmentov pôvodného telesa. Tie preberajú materiál taký, aký malo pôvodné teleso.

Ako bolo popísané v kapitole 2.1.1, telesá môžu byť dvoch typov - statické a dynamické. Všetky úlomky získajú dynamický typ, zatiaľ čo fragmenty pôvodného telesa prevezmú typ od primárneho telesa, z ktorého vznikli.

Je tiež nutné ošetriť fragmenty extrémnych tvarov. Vznik telies, ktorých hmotnosť je nižšia, ako určitý limit, sú pre simuláciu zanedbateľné. Preto sa ich ignorovaním zvýši výkon bez výrazných následkov na jej kvalitu.

Najdôležitejší atribút, ktorý je nutné definovať, je pohyb. Engine po náraze predpokladá, že sa telesá zachovajú a preto im na základe toho vypočíta novú rýchlosť otáčania a vektor pohybu. Tu je nutné definovať účinok kolízie:

$$z = \frac{p}{i}$$

kde p je pevnosť telesa a i je intenzita kolízie. Vieme, že $i \geq p \rightarrow z \in \langle 0, 1 \rangle$. Vďaka miernej modifikácii kódu JBox2D je možné pristupovať k starým hodnotám uhlovej rýchlosti ω a vektoru pohybu v. Označme staré hodnoty indexom s a nové indexom n. Reálne hodnoty vypočítame:

$$v = v_s + z(v_n - v_s)$$
$$\omega = \omega_s + z(\omega_n - \omega_s)$$

Pokiaľ $z = 1$, intenzita je rovná pevnosti a môžeme tvrdiť, že všetka hybnosť bola spotrebovaná na rozbitie telesa, teda výsledkom bude v_n. V opačnom prípade $z = \epsilon$ nastalo rozbitie bez prakticky žiadnej námahy a teda logicky platí, že kontakt medzi bodmi je ignorovaný, na základe čoho je telesám nastavený ich pôvodný vektor (Príklad: pád pieskového telesa). Zvyšné hodnoty sú počítané lineárnym prechodom medzi týmito dvomi vektormi.

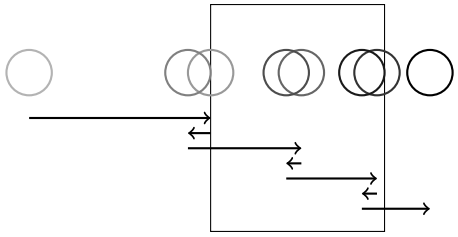
V prípade, že sekundárne teleso nepodlieha triešteniu, rovnakým postupom upravíme atribúty aj jemu. Ak triešteniu podlieha, o túto úpravu sa postará fraktúra, v ktorej reprezentuje primárne teleso.

Všetkým fragmentom nastavíme uhlovú rýchlosť ω. Nový vektor pohybu fragmentu zistíme tak, že podľa hodnoty ω a vektoru pohybu v vypočítame pohyb ťažiska fragmentu vo svetových súradniciach:

$$v_f = \left| \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} (t_f - t) \right| \omega + v$$

kde t_f je ťažisko fragmentu a t je ťažisko primárneho telesa vo svetových súradniciach.

Pri kolidovaní telies okrem iného vzniká problém posunu. Kvôli diskretizácii simulácie sa telesá pri ich vzájomnej kolízii do seba zanorujú o určitý interval, kvôli čomu engine dané telesá transformuje o mieru vzájomného zanorenia a umelo ich „vysunie“ od seba. To môže mať pri simulácii projektilu nežiaduce účinky na vizuálnu stránku kvôli trhanému pohybu strely. To je možné ošetriť tým, že sa sekundárnemu telesu nastaví jeho pôvodná transformácia (poloha) a podľa vypočítaných hodnôt vektoru pohybu a dĺžke frame-u sa vypočíta poloha nová. Tento prístup zaručuje prirodzené chovanie projektilu pri priestrele objektu.



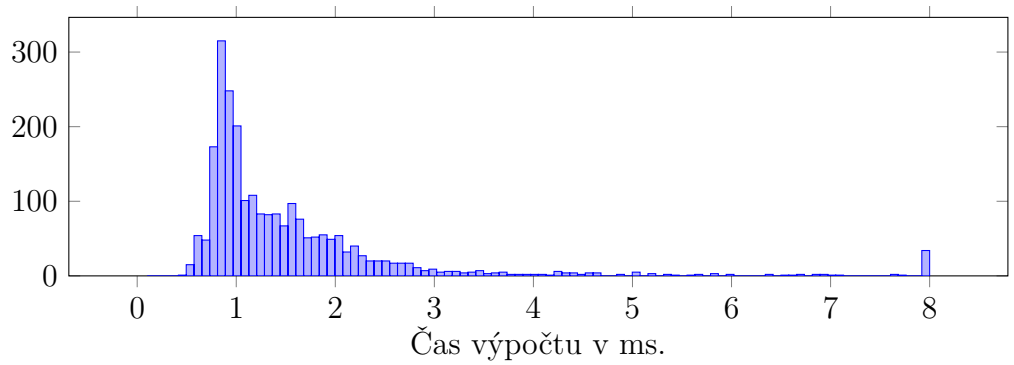
Obr. 6.4: Pohyb projektilu pri neošetrení kolíznej transformácie

6.7.4 FractureListener

Interface *org.jbox2d.fracture.FractureListener* dáva programátorovi možnosť nadefinovať kód, ktorý sa zavolá vždy pri nastaní fraktúry.

- **`FractureListener`** Interface deklarujúci metódu, ktorá sa spustí vždy po spracovaní fraktúry.
  - **`void action()`** Handler udalosti. Má atribúty:
    - **`Material material`** Materiál rozpadajúceho sa telesa.
    - **`float intensity`** Intenzita kolízie.
    - **`List<Body> fragments`** Telesá fragmentov.

6.7.5 Časová náročnosť



Obr. 6.5: Histogram časov výpočtu trieštenia

Uvedený histogram zobrazuje početnosť časov strávených na kompletnom procese trieštenia (vygenerovanie voroného diagramu, fragmentácia, konvexná dekompozícia fragmentov, vloženie nových telies do simulácie) pre 2500 vzoriek. Do úvahy treba brať aj štatistickú odchýlku spôsobenú volaním *garbagge collectora* počas vykonávania daného procesu, čo má za následok výrazné predĺženie doby výpočtu.

7. Materiály

V tejto kapitole sa budeme zaoberať spôsobom, akým je možné nadefinovať trieštenie objektu a nastaviť mu príslušné vlastnosti. Ako bolo spomenuté v kapitole 6.2, nástroje poskytujúce tieto možnosti sú implementované v triede *Material*.

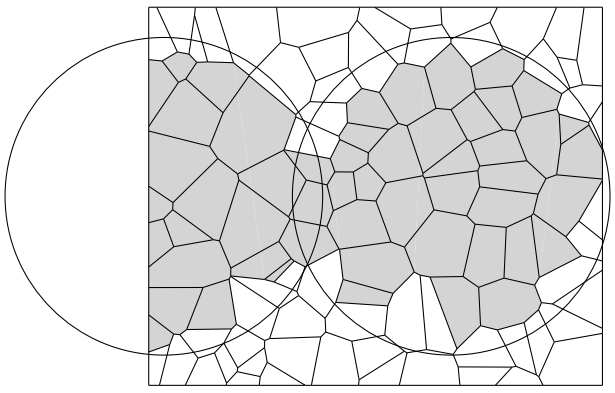
- **`Material`** Abstraktná trieda definujúca materiál.
  - **`float m_rigidity`** Definuje pevnosť telesa. Ak nastane kolízia, ktorej *momentInertia* presahuje túto hodnotu, nastane fraktúra.
  - **`float m_shattering`** Trieštivosť. Určuje relatívnu veľkosť fragmentov.
  - **`float m_radius`** Nastavuje polomer pre filter hranice účinnosti (viď kapitola 5.2.2).
  - **`Material m_fragments`** Určuje materiál úlomkov.
  - **`Vec2[] focee()`** Abstraktná metóda definujúca spôsob trieštenia materiálu. Generuje ohniská voroného diagramu, čím určuje tvary fragmentov. Má parametre:
    - **`Vec2 contactPoints`** Kolízny bod.
    - **`Vec2 contactVector`** Vektor nárazu.

  Implementácie triedy *Material* sú umiestnené v priečinku *materials*, ktorý obsahuje nasledujúce triedy:

- **`Glass`** Sklo.
- **`Diffusion`** Materiál s logaritmickým rozptylom ohnísk.
- **`Uniform`** Materiál s rovnomerným rozptylom ohnísk.

7.1 Terminálna balistika

Špeciálnym prípadom simulácie je dopad projektilu na teleso, pri ktorom vzniká problém diskretizácie výberu fragmentov počas jednotlivých frame-ov. Na riešenie tohto problému stojí za úvahu myšlienka nastavenia hodnoty $\|v\|$ z kapitoly 5.2.2 podľa odhadu hĺbky prieniku.



Obr. 7.1: Diskretizácia výberu fragmentov pre $\|v\| = r$

Na vytvorenie tohto odhadu slúži *terminálna balistika*, ktorou sa zaoberá John A. Zook v knihe *Terminal ballistics test and analysis guidelines for the penetration mechanics branch* [25], kde okrem iného popisuje atribúty, ktoré hĺbku prieniku projektilu ovplyvňujú, ktoré sú: typ materiálu, hustota materiálu, hmotnosť a tvar projektilu. Vieme, že medzi najdôležitejšie aspekty patrí práve tvar projektilu. Fyzikálne procesy prebiehajúce pri prieniku projektilu telesom sú však veľmi zložité a z pohľadu hernej simulácie je preto veľmi náročné tento aspekt predikovať. Je preto lepšie túto úlohu nechať na viac krokov simulácie. Definovať tento atribút je preto vhodné iba na základe testovania a pozorovania, v ktorých prípadoch sa simulácia javí najprirodzenejšie. Pri testovaní sa ukázalo, že dobré výsledky dáva

$$\|v\| = max(cd, r)$$

kde c je konštanta ($c \geq 1$) a d je dĺžka dráhy, ktorú projektil urazí medzi frameami.

7.2 Generátor ohnísk

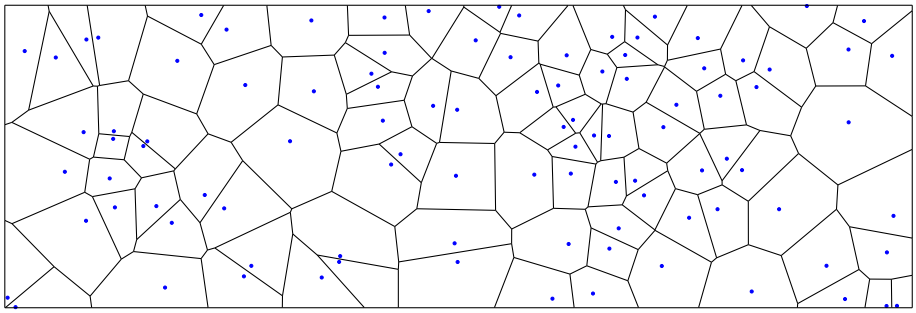
Základná myšlienka definície materiálov spočíva v implementovaní abstraktnej metódy *focee()* deklarovanej v objekte *Material* generujúcej ohniská voroného diagramu. V projekte je implementovaných niekoľko materiálov, pričom každý materiál je založený na špecifickom generovaní ohnísk.

Označme si k ako počet generovaných ohnísk, kde s je trieštivosť, d kolízny bod a v vektor pohybu. Definujme si funkciu $p()$, ktorá vráti pseudonáhodné číslo v intervale $\langle -\frac{1}{2}, \frac{1}{2} \rangle$. Pomocou uvedených hodnôt je možné definovať niekoľko nasledujúcich materiálov.

7.2.1 Rovnomerný rozptyl

Je generované náhodnými bodmi rovnomerne rozmiestnenými na ploche. Ohniská je možné vypočítať pomocou:

$$v_i = (p(), p())s\sqrt{k} + d$$



Obr. 7.2: Rovnomerný rozptyl

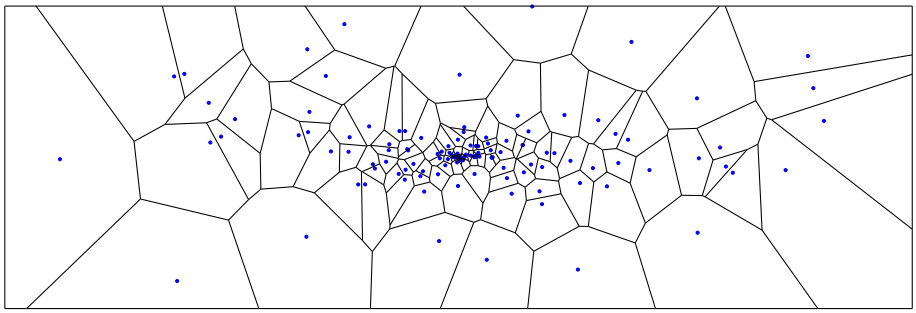
7.2.2 Logaritmický rozptyl

Ďalšia možnosť je generovať ohniská s logaritmickým rozptylom v polárnych súradniciach s väčšou hustotou v bode nárazu. Tento návrh vznikol z predpokladu,

že pri trieštení telesa budú v okolí kolízneho bodu menšie úlomky ako na periférii fragmentácie. Body sú generované:

$$\begin{aligned} \alpha &= p()2\pi \\ r &= -\ln r()\, s \\ v_i &= T(\sin\alpha\, r, \cos\alpha\, rc) \end{aligned}$$

kde c je konštanta určujúca predĺženie rozptylu v smere vektora v a T je transformačná matica popísaná v kapitole 5.2.2.

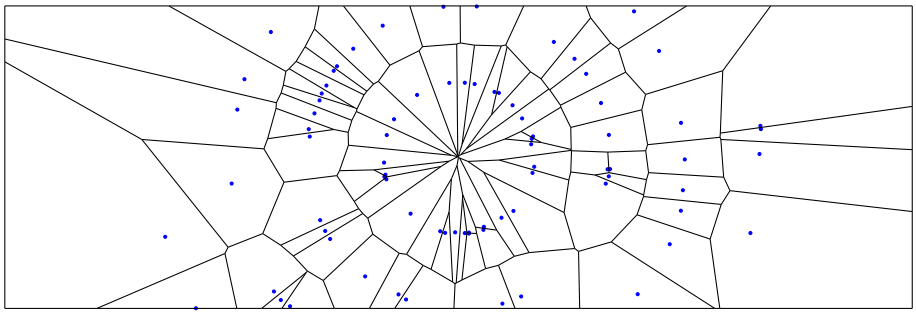


Obr. 7.3: Logaritmický rozptyl

7.2.3 Sklo

Definícia skla sa opäť týka polárnych súradníc. Body sú generované na niekoľkých kružniciach s malými odchýlkami. Označme n ako počet kružníc. Body definujeme ako:

$$\begin{aligned} \alpha &= p()2\pi && \text{(náhodný uhol)} \\ \xi &= p()sc && \text{(odchýlka)} \\ r_i &= is + \xi && \text{(polomer)} \\ v_{i,k} &= (\sin\alpha\, r_i, \cos\alpha\, r_i) + d && 1 \leq i \leq n,\ i \in \mathbb{N} \end{aligned}$$

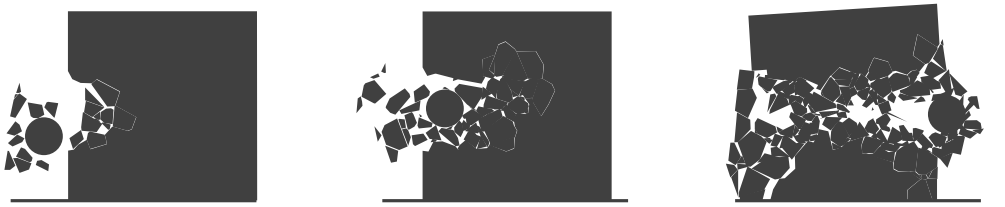


Obr. 7.4: Kružnicové rozloženie

c je konštanta určujúca mieru rozptylu bodov od kružníc.

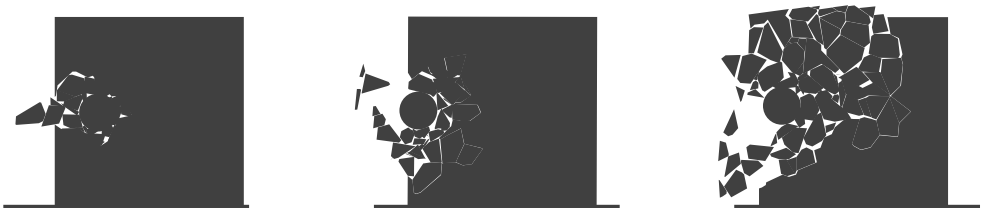
7.3 Testy vlastností materiálov

V uvedenej sekcii máme k dispozícii testy správania objektov pri rôznom nastavení atribútov *m_rigidity*, *m_shattering* a účinnosti *m_radius*.



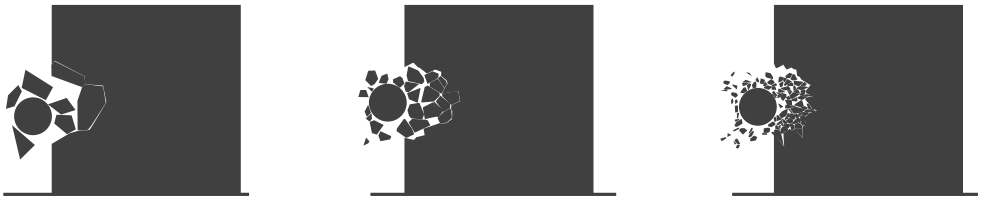
(a) m_rigidity = 100 (b) m_rigidity = 50 (c) m_rigidity = 20

Obr. 7.5: Správanie telies pri rôznej pevnosti materiálov



(a) m_rarius = 1 (b) m_radius = 2 (c) m_radius = 5

Obr. 7.6: Správanie telies pri rôznom polomere účinnosti



(a) m_shattering = 10 (b) m_shattering = 4 (c) m_shattering = 1,5

Obr. 7.7: Správanie telies pri rôznej miere trieštivosti

Na uvedené testy bol aplikovaný materiál rovnomerného rozptylu. Je tiež možné skúmať proces trieštenia pri kombinácii ďalších parametrov, ako napr. rôzne tvary telies, hustota, hladkosť povrchu alebo odrazivosť, ktoré sú súčasťou štandardnej knižnice JBox2D a majú na daný proces značný vplyv.

8. Rozhranie

Uvedená kapitola poskytuje popis programátorského rozhrania pre využitie implementovaných algoritmov.

8.1 Vytvorenie nekonvexného telesa

Implementácia poskytuje možnosť vytvoriť nekonvexný polygón v simulačnom prostredí:

```
PolygonFixture pol = new PolygonFixture(new Vec2[] {
     new Vec2(1, 2),
     ...
});
body.createFixture(def, pol);
```

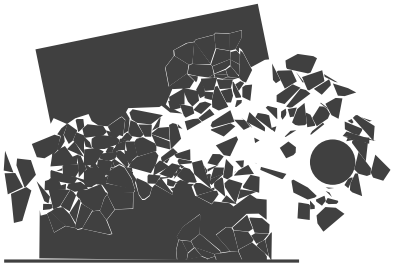
Na začiatku sme vytvorili nový polygón *pol*, ktorý sme následne pridali do telesa *body* typu *Body* pomocou metódy *createFixture()*. Tá nie je súčasťou štandardnej knižnice JBox2D a bola pridaná do triedy *Body* kvôli zachovaniu pôvodnej logiky. Metóda na daný polygón aplikuje konvexnú dekompozíciu a podľa šablóny *def* typu *FixtureDef* vloží do telesa *body* príslušné predmetys danými vlastnosťami.

8.2 Nastavenie materiálu

Materiál je reprezentovaný atribútom v predmete (*Fixture*). Spôsob jeho nastavovania je založený na rovnakom princípe, ako nastavenie akéhokoľvek iného atribútu daného objektu. Je to možné priamo nastavením premennej *fixture.m_material*, alebo pomocou šablóny *FixtureDef* nastavením hodnoty *fixtureDef.material*. Inicializovať materiál je možné konštruktorom jednotlivých implementácií z balíka *org.jbox2d.fracture.materials*, alebo priradením hodnôt zo statických premenných triedy *Material*. Daným materiálom je následne možné nastavovať atribúty *m_rigidity*, *m_shattering*, *m_radius* a *m_fragments* (viď kapitola 7).

V scéne definovanej v kapitole 2.2 môžeme predmetu telesa *cube* nastaviť materiál rovnomerného rozptylu príkazom:

```
f2.m_material = Material.UNIFORM;
```



Obr. 8.1: Vizualizácia úvodnej scény s podporou fraktúr

9. Záver

Cieľom práce bola podpora trieštenia telies po ich kolízii a simulovanie rôznych materiálov s rôznymi vlastnosťami v grafickom užívateľskom rozhraní s dôrazom kladeným na nízke časové nároky, prirodzené chovanie daných procesov a jednoduché programátorské rozhranie zachovávajúce logiku pôvodnej knižnice. Na základe výsledkov testov a meraní výkonu môžeme konštatovať, že tento cieľ bol splnený.

V práci je možné nadväzovať viacerými spôsobmi, ktoré do nej z časových dôvodov zahrnuté neboli, ako napr. podpora vkladania polygónov s dierami, explózie, tlakové vlny, alebo podpora trieštenia objektov na základe viacerých kolíznych bodov. Je tiež možné bližšie študovať správanie materiálov v 2D pomocou definovania rôznych kombinácií vlastností, ktoré poskytnú široké možnosti pri simuláciách a vývoji hier.

Implementácia je napísaná v jazyku *Java* [5], čo umožňuje vývoj aplikácií pre zariadenia s operačným systémom *Android* [7], desktopových aplikácií nezávislých na platforme, alebo aplikácií využívajúcich technológiu *Java web start*.

Zoznam použitej literatúry

[1] Box2D. http://box2d.org/.

[2] JBox2D. http://www.jbox2d.org/.

[3] poly2tri. https://code.google.com/p/poly2tri/.

[4] Liquid Fun. http://google.github.io/liquidfun/.

[5] Java. http://java.com/.

[6] NetBeans. https://netbeans.org/.

[7] Android. http://www.android.com/.

[8] Regents of the University of California. *New BSD License.* Public Domain, July 1999.

[9] Gailly, J. - Adler, M. *Zlib License.* March 1995.

[10] Seymour, M. *Art of Destruction (or Art of Blowing Crap Up).* [online], 2011. Dostupné z: http://www.fxguide.com/featured/art-of-destruction-or-art-of-blowing-crap-up/.

[11] Lee, D. - Schachter, B. *Two Algorithms for Construction a Delaunay Triangulation.* International Journal of Computer and Information Science, vol. 9, No. 3. 1980.

[12] Dyer, R. - Zhang, H. - Moller, T. *Observations on Gabriel meshes and Delaunay edge flips.* GrUVi Lab, School of Computing Science, Simon Fraser University, Canada, 2008.

[13] Aichholzer, O. - Mulzer, W. - Pilz, A. *Flip Distance Between Triangulations of a Simple Polygon is NP-Complete.* EuroCG 2013, Braunschweig, Germany, March, 2013.

[14] Lawson, C. *Transforming triangulations.* Discrete Math., 3(4):365-372, 1972.

[15] Sleator, D. - Tarjan, R. - Thurston, W. *Rotation distance, triangulations and hyperbolic geometry.* J. Amer. Math. Soc., 1:647-682, 1988.

[16] Aurenhammer, F. - Klein, R. - Lee, D. *Voronoi Diagrams and Delaunay Triangulations.* World Scientific, 2013, ISBN 978-9814447638.

[17] Fortune, S. *A sweepline algorithm for Voronoi diagrams.* Proceedings of the second annual symposium on Computational geometry. Yorktown Heights, New York, US, 313–322, 1986. ISBN 0-89791-194-6.

[18] O'Rourke, J. *Computational Gemoetry In C.* Second Edition. Cambridge University Press, 44-62, 1997.

[19] Žára, J. *Počítačová grafika - principy a algoritmy.* Grada a. s., 73-89, 1992. ISBN: 80-85623-00-5.

[20] Seidel, R. *A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulation Polygons.* Computer Science Division, University of California Berkley CA 94720, USA, 1991.

[21] Chazelle, B. - Dobkin, D. *Optimal convex decompositions.* Computational Geometry, G. T. Toussaint, North Holland, Amsterdam, 63-133, 1985.

[22] Hertel, S. - Mehlhorn, K. *Fast triangulation of a simple polygon.* Lecture Notes in Computer Science, vol. 158, Berlin/New York, 207-218, 1983.

[23] Finley, D. - Mullen, P. *Point-In-Polygon Algorithm - Determining Whether A Point Is Inside A Complex Polygon.* [online], 2007. Dostupné z: `http://alienryderflex.com/polygon/`.

[24] Bloch, J. *Effective Java.* Second Edition. Addison Wesley, 2008. ISBN: 978-0-321-35668-0.

[25] Zook, J. - Frank, K. - Silsby, G. *Terminal Ballistics test and analysis guidelines for the penetration mechanics branch.* Ballistics Research Laboratory, Aberdeen proving ground, Maryland, 13-14, 1992.

Prílohy

A. Obsah CD

Priložené CD obsahuje nasledujúce súbory:

- **src** - priečinok so zdrojovými kódmi
- **gui.jar** - spustiteľný súbor vizualizujúci testovacie scenáre
- **thesis.pdf** - text bakalárskej práce

Zdrojové kódy je možné zakompilovať pomocou vývojového prostredia *Netbeans IDE* verzie 8 [6].

B. GUI

Grafické užívateľské rozhranie pre podporu vizualizácie je implementované v priečinku *org/gui*. Poskytuje spustiteľný súbor *Tests.java* a sadu testovacích scenárov v priečinku *testbed*. Po jeho spustení sa otvorí okno s jedným canvas-om určeným na vykresľovanie simulácie a combobox-om umožňujúcim výberu testovacieho scenára. Ovládanie aplikácie je nasledovné:

- s - štart/stop
- r - nastaví simuláciu do počiatočného stavu
- pravé tlačidlo myši - posúvanie kamery
- ľavé tlačidlo myši - pohybovanie dynamickými objektmi
- koliesko myši - približovanie/vzďaľovanie obrazu

Počas simulácie sa na ľavej strane canvas-u zobrazujú:

- informácie o ovládaní:
  - klávesové ovládanie
  - pozícia myši
  - pozícia stredu obrazu
  - zoom
- informácie o simulácii:
  - počet telies (Bodies)
  - počet predmetov (Fixtures)
  - počet kontaktov (Contacts)
  - počet partícií kvapaliny (Particles)