Önálló laboratórium (BMEVIIIAL01) beszámoló

Téma: Position Based Dynamics

(ruhadarab szimulációja)

Hallgató neve: Pecze Dávid

Neptun: PPV33A

Konzulens: Dr. Szécsi László

1. Bevezetés

Általában fizikai szimulációkban számontartjuk a tömegeket, és az erőket. Ami a bonyodalmat okozhatja és okozza is, az a második derivált, azaz a gyorsulás számítása. Minél bonyolultabb kényszereket fogalmazunk meg, annál komplikáltabb és számításigényesebb lesz a szimuláció. A Position Based Dynamics ekörül próbál eljárni, nem feltétlenül megoldani, hanem kellően megközelíteni. Majdnem olyan egyszerű, mint egy Explicit Euler integrálás és majdnem olyan stabil, mint Implicit Euler. A szimulált részecskékben nem tartunk számon erőket, hanem:

- aktuális pozíciójukat x
- sebességüket v
- inverz tömegüket w
- jövőbeli pozíciójukat p

Majd minden lépésben ezekkel az adatokkal dolgozva számolunk jövőbeli pozíciót az ezekre a részecskékre megfogalmazott kényszerek vetítésével. Egy kényszerhez tartozik valahány részecske, egy stiffness ∈ [0…1] érték, ami megmondja az erejét és ha úgy vesszük, része még egy függvény is. Minden időlépésben adott iteráció-számszor vetítjük a kényszereket. Minél nagyobb ez a szám, annál pontosabb a szimuláció, de annál lassabb is. Az alap algoritmus pszeudó-kódja:

forall vertices *i*
    initialize $\mathbf{x}_i = \mathbf{x}_i^0$, $\mathbf{v}_i = \mathbf{v}_i^0$, $w_i = 1/m_i$
endfor
loop
    forall vertices *i* **do** $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t w_i \mathbf{f}_{ext}(\mathbf{x}_i)$
    dampVelocities($\mathbf{v}_1, \ldots, \mathbf{v}_N$)
    forall vertices *i* **do** $\mathbf{p}_i \leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$
    forall vertices *i* **do** generateCollisionConstrains($\mathbf{x}_i \rightarrow \mathbf{p}_i$)
    loop solverIteration **times**
        projectConstraints(C_1,..., $C_{M+M_{coll}}$, $\mathbf{p}_1$,..., $\mathbf{p}_N$)
    endloop
    forall vertices *i*
        $\mathbf{v}_i \leftarrow (\mathbf{p}_i - \mathbf{x}_i) / \Delta t$
        $\mathbf{x}_i \leftarrow \mathbf{p}_i$
    endfor
    velocityUpdate($\mathbf{v}_1, \ldots, \mathbf{v}_N$)
endloop

2. Környezet

Több lehetőségem is volt arra, hogy miben implementáljam a szimulációt. Lehetett volna létező játékmotorban, mint például *Unreal Engine* vagy *Unity*. Amellett döntöttem, hogy én magam írom meg a 3D megjelenítő motort, ezzel is gyakorolva a grafikai programozást. A választott programnyelv az **C++**. Input és ablak kezelésére a **GLFW** könyvtárat használtam, grafikus API-nak, pedig az **OpenGL**-t választottam. A videókártya programozására **CUDA**-t használtam. Az ablakon adatok megjelenítéséhez, interaktív elemek használatához az **ImGui** C++ könyvtárat találtam a legmegfelelőbbnek. Vektor könyvtárnak a **glm**-re esett a választás.

Árnyaláshoz a **Phong-Blinn** megvilágítási modellt- követtem. **Irány** és **pontfényforrást**, valamint **fényszóró** jellegű fényforrást is használtam a projekt során.

3.1 Kezdetek

Első célom az volt, hogy egy **ingát** csináljak. Ez az egyik legegyszerűbb objektum, amit meg lehet fogalmazni egy ilyen jellegű szimulációban. Ebben az esetben **8 részecskéből** áll és minden szomszédos részecske között meg van fogalmazva egy **távolság kényszer**. Ez nagyon hasonlít egy rugószerű kapcsolathoz két test/pont között. A távolság kényszer próbálja úgy mozgatni a hozzá tartozó 2 részecskét, hogy a kettő közötti távolság, az a létrehozásánál beállított távolság legyen. Az, hogy mekkora mértékkel változtatja meg a pozíciót egy lépésben, az **függ** a **nyugalmi távolságtól** való aktuális **eltéréstől** és a **stiffness** értékétől. Minél nagyobb, annál inkább meg akarja tartani az alaptávolságot. Ez mind a 2 értékre igaz.

Eredmény:

1 iteráció 10 iteráció 20 iteráció

- Felső sor: stiffness = 1
- Alsó sor: stiffness = 0.01

Minden részecske helyén egy gömb van. Ami részecskék között távolság kényszer áll fent, azok össze vannak kötve. Látszik, hogy minél kevesebbszer vetítjük a kényszereket (azaz minél

kisebb az iterációk száma, annál „nyúlósabb”, annál jobban „szétfolyik” az objektum. Ha nagyobbra állítjuk a számot, akkor pontosabb lesz, de időben tovább is fog tartani a számítás.

3.2 Inga folytatása

Úgy haladtam tovább, hogy lekezeltem a kódban, az **ütközést** egy **gömbbel**. A gömb statikus objektum, ütközéskor csak a részecskék sebessége változik. Az inga most egy kötélként van ábrázolva, oly módon, hogy vastagabb, feltextúrázott hengerrel vannak összekötve a részecskék, gömb pedig nincs kirajzolva a helyükre. Textúraként egy folytonos (seamless) kötél textúrát választottam.

Ebben a verzióban az **újdonság** az **ütközés detektálás**. Egyelőre csak **részecske-gömb** ütközést nézek. Nem a gömb háromszöghálójához képest nézem, hanem a gömb geometriai reprezentációjához. Ez azt jelenti, hogy nem egyenként nézem meg a háromszögeket, hogy ütközött-e velük, hanem a középponttól való távolságot ellenőrzöm. 2 eset lehetséges.

(1) Még kívül van, de látjuk, hogy a következő iterációban bele fog lépni

(2) Már belépett a gömbbe.

Ha még nem ütközött, de detektáltuk, hogy fog, akkor a következő módon járunk el (1):

Kiszámoljuk a belépési pontot és felületi normált abban a pontban. Ezek segítségével egy olyan kényszert adunk hozzá a kényszerek listájához, ami az érintett részecskét a felületi normál irányába tolja el (tehát elfele a testtől, amivel ütközik). Az eltolás mértéke attól függ, mennyire fog behatolni a következő iterációban. A függvény a következőképpen néz ki:

$C(\mathbf{p}) = (\boldsymbol{p} - q_c) \cdot \boldsymbol{n_c}$, ahol C a kényszer, **p** a részecske jövőbeli pozíciója, $\boldsymbol{q_c}$ a belépési pont és $\boldsymbol{n_c}$ a felületi normál abban a pontban.

Ha már teljesen benne van a testben, akkor hasonló módon cselekszünk (2):

Megkeressük a legközelebbi pontot **p**-hez a felszínen, ez lesz $\boldsymbol{q_s}$, szintén kiszámoljuk ott a felületi normált, $\boldsymbol{n_s}$. A függvény most ugyan az, csak ezekkel az adatokkal:

$C(\mathbf{p}) = (\boldsymbol{p} - \boldsymbol{q_s}) \cdot \boldsymbol{n_s}$

Mindkét esetben a stiffness értéke 1. Súrlódással nem foglalkoztam.

Használt egyenletek:

Sugár: $\text{Ray}(t) = \text{ray.start} + t \cdot \text{ray.dir}$

Gömb: $|r - c|^2 = R^2$

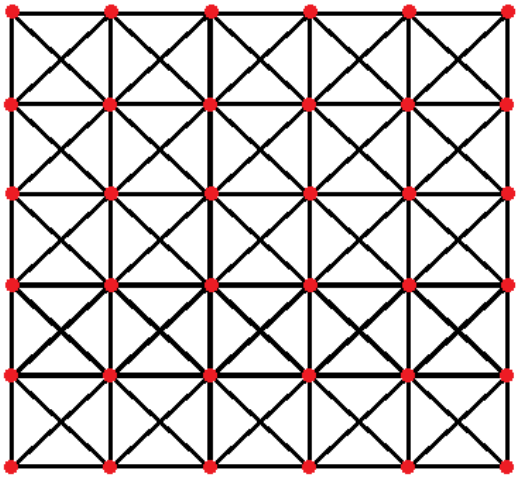
Eredmény:



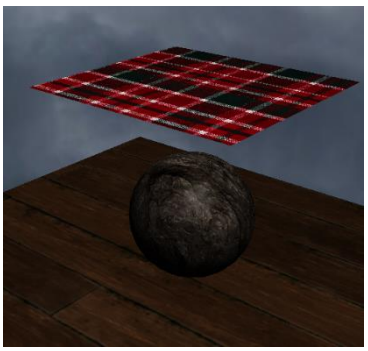
3.3 Ruhadarab CPU-n

A következő logikus lépés az, hogy egy ruhadarabot szimuláljak. Ennek a reprezentációja a következő:

Egy sík mentén generálok egy **2D**-s **rácsot**. A rács négyzeteinek csúcspontja egy-egy **részecske**, a rács minden vonala egy **távolság kényszer**, plusz még átlósan is össze vannak kötve. A képen minden piros pont egy részecske és minden részecskét összekötő fekete vonal egy kényszert jelöl. Lehetne minden szomszédos háromszög között egy **hajlási kényszert** megadni, de nem muszáj, anélkül is megfelelő a szimuláció és gyorsabb is. A megjelenítése egyértelmű, **minden részecske egy Vertex**. Az ütközés detektálás ugyanúgy működik, mint eddig.



Eredmény:



50x50 mesh, 64 iteráció, stiffness = 1, minden PBD-hez kapcsolódó számítás a CPU-n történik.

Kb. 90-100 FPS Intel Core i5–6500 processzoron.

3.4 Ruhadarab GPU-n

A következő lépés az volt, hogy a **távolság** és **ütközés** kényszerek vetítését átírtam úgy, hogy a **videókártyán** legyen számolva. Ez nem egy triviális feladat. CUDA-t használva a következő módon oldottam meg a feladatot.

Az ütközési kényszereket, mivel **minden időlépésben** újra kellett generálni, muszáj voltam oda-vissza másolgatni a processzor és a videókártya memóriája között.

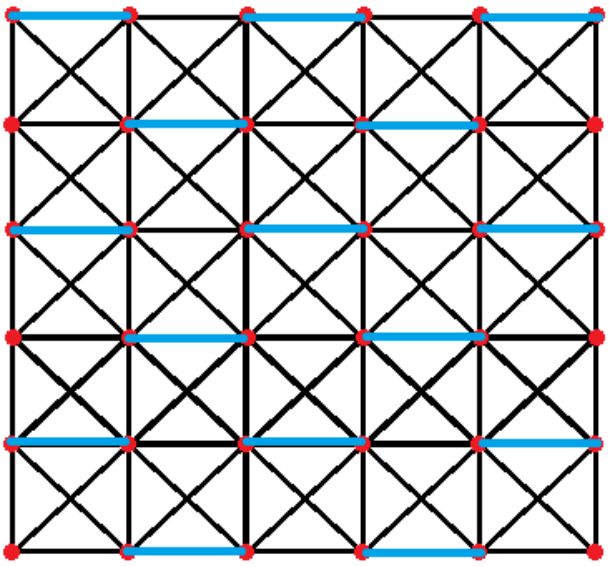
A távolság kényszereket elég csak a program **futása elején** feltölteni a GPU memóriába, nem változnak. Ha mondjuk lenne szakadás, akkor változnának. A bonyodalmat az jelenti, hogy nem lehet párhuzamosan számolni az összes kényszert, hiszen vannak közös pontjaik. A feladat az, hogy olyan **csoportokra** bontsuk, amikben **nincs** 2 olyan kényszer, ami tartalmazza ugyanazt a pontot. Ennek az általános megoldása az lenne, hogy veszünk egy olyan irányítatlan gráfot, aminek minden csúcsa egy kényszer, két csúcs pedig akkor van összekötve, ha van közös pontjuk. Ha futtatunk egy gráfszínező algoritmust, akkor minden azonos színű csúcs(kényszer) egy csoportba tartozik, mivel nincs közös pontjuk.

A ruhadarab esetén nem muszáj ezt implementálni, ott magunktól is létrehozhatjuk a csoportokat, nem olyan bonyolult. Kezdjük el felbontani őket. **4 csoportba** úgy tudjuk szétosztani őket, hogy „irányuk” alapján csoportosítunk:

- (U) -val párhuzamos
- (V) -vel párhuzamos
- (U+V) -vel párhuzamos
- (U–V) -vel párhuzamos

Ahol **U** és **V** a **textúrakoordinátarendszerben** értelmezett **bázisvektorok**. (A párhuzamosság a nyugalmi állapotban értendő.)

Ezeket tovább kell osztani, mivel, ha vesszük például az összes U-val párhuzamos kényszert, akkor minden pont, az elsőt és utolsót kivéve, 2 kényszerben van benne. Emiatt mondjuk egy csoportban **minden páratlan** sorszámú, **U-val párhuzamos** kényszer van. A képen ezek vannak ábrázolva. (kék vonalak) Látszik, hogy nem fér el több kényszer, nem tudunk úgy hozzáadni, hogy ne érintkezzen a meglévőkkel. Ha a felső 4 csoportot felosztjuk még 2-2-re, akkor összesen 8-at kapunk. Ez a végeredmény. **8 menetben tudjuk** őket, menetenként **párhuzamosan** vetíteni.



Ütközés kényszereknél nem volt ilyen probléma, mivel csak 1db részecskére vonatkozik. Ott egy egyszerű, párhuzamosított bejárást végezhetünk.

Megfigyelések:

- A teljesítmény akkor volt a legjobb, ha a blokk mérete 32 vagy 64 volt. Ha több vagy kevesebb volt, akkor romlott a sebesség.
- Ugyan maguk a kényszerek, legalábbis a távolságra vonatkozók, csak egyszer voltak feltöltve a GPU-ra, a részecskéket minden időbeli lépésben **oda-vissza** kell másolni a memóriából. Ezen lehet optimalizálni, úgy, hogy minél inkább a GPU-n tartjuk az adatot. Például a **VAO**-t össze lehet kötni CUDA **device-pointerrel**.

Ezek implementálása után az **50x50** mesh, **64** iterációval **200 FPS**-el futott. Volt még valami, amin lehetett optimalizálni, mégpedig a tárolási struktúráján a részecskéknek. Eddig úgy tároltam őket, hogy volt egy **MassPoint** osztályom, amiből csináltam egy tömböt. Teljesítmény szempontjából előnyösebb fordítva gondolkodni és egy olyan osztályt írni, ami az összes részecskét képviseli, és egyenként tartalmaz tömböket.

Array of Structure(Régi)

```
struct MassPoint {
        glm::vec3 x;
        glm::vec3 v;
        float w;
        glm::vec3 p;
};

MassPoint* massPoints =
new MassPoints[n];
```

Structure of Arrays (Új)

```
struct MassPoints {
        glm::vec3* x;
        glm::vec3* v;
        float* w;
        glm::vec3* p;
};

MassPoints massPoints;
massPoints.x = new glm::vec3[n];
//...
```

Az új módszer előnyösebb, mivel **nem szakadozva** érhetőek el, az egyes adatok. Gyakoribb az, hogy mondjuk csak 2-re vagyok kíváncsi, nem mind a 4-re. Eddig muszáj voltam mindig az **egész tömbön** végigmenni és kiolvasni mindent, akkor is, ha mondjuk csak a sebességre és pozícióra voltam kíváncsi. Így most megtehetem azt, hogy külön átadom csak azt a 2 tömböt, és megszakítás nélkül végigmegyek rajtuk. Bevált az ötlet, megint gyorsabb lett a program. **100x100**-as mesh esetén **24** iterációval időlépésenként, kb. **280-300** FPS volt, ami kiváló javulást jelent.

3.5 Ön-ütközés

A félév végén egy dolgot írtam még meg, ami nem más, mint a részecskék ütközése már részecskékkel. Ennek detektálása és lekezelése mind CPU-n történik, így a teljesítmény visszaesett. Itt mind a kettő dolog trükkös, egyrészt nem elfogadható az, hogy minden részecskéhez megnézek minden másikat, hiszen az $O(n^2)$, másrészt nem mindegy hogyan kezelem az ütközést, miután detektáltam.

Az ütközés detektálásához **spatial hashing**-et alkalmaztam. Előnye, hogy nem egy véges térben értelmezendő, hanem elméletben végtelenre, a hash függvény modulo természete miatt. Amit tárolok, az csak az index, nem a konkrét adatok. A **3D** térből **1D** tömbbe leképző függvény:

```
int toHashIndex(int x, int y, int z) {
        int h = (x * 92836111) ^ (y * 689286499) ^ (z * 283923481);
        return abs(h) % tableSize;
}
```

A függvényben alkalmazott számokat Matthias Müller-Fischer találta ki, de ezeket szabadon lehet használni. Egy **cellának** a **méretét** érdemes akkorára választani, amekkora a **nyugalmi távolsága** 2 részecskének a ruhában. A tábla méretét a részecskék számának 10-szeresére állítottam be, ezt volt a leggyorsabb konfiguráció. A hash táblát minden időlépésben legenerálom az aktuális pozíciókkal. Minden részecskéhez lekérdezem a közeli részecskék indexét, és köztük nézek ütközést. Közeli jelen esetben a nyugalmi távolságra lévőket jelenti.

Ha detektáltam ütközést, akkor röviden, annyit csinálok, hogy egymástól ellentétes irányba eltolom őket, egyenlő mértékkel. Súrlódást úgy lehet elérni, hogy a sebességüktől függően korrekciók végzünk az eltoláson, a következő módon: (friction ∈ [0…1])

$\boldsymbol{x}_1 \leftarrow \boldsymbol{x}_1 + \text{friction} \cdot (\boldsymbol{v}_{avg} - \boldsymbol{v}_1)$

$\boldsymbol{x}_2 \leftarrow \boldsymbol{x}_2 + \text{friction} \cdot (\boldsymbol{v}_{avg} - \boldsymbol{v}_2)$

Mivel az átlagsebesség a következő módon jön ki:

$\boldsymbol{v}_{avg} = (\boldsymbol{v}_1 + \boldsymbol{v}_2) / 2$

Ezért az összsebesség marad, és arányosan csak egy részével dolgozunk. Ha behelyettesítjük:

$\boldsymbol{v}_{avg} - \boldsymbol{v}_1 + \boldsymbol{v}_{avg} - \boldsymbol{v}_2 = 2 \cdot \boldsymbol{v}_{avg} - \boldsymbol{v}_1 - \boldsymbol{v}_2 = 2 \cdot (\boldsymbol{v}_1 + \boldsymbol{v}_2) / 2 - \boldsymbol{v}_1 - \boldsymbol{v}_2 = \boldsymbol{v}_1 + \boldsymbol{v}_2 - \boldsymbol{v}_1 - \boldsymbol{v}_2 = 0$

4. Felhasználói interakció

Kiegészítésként megírtam, hogy egérrel lehessen mozgatni a ruhát. Pontosabban egy adott **részecske pozícióját** manipulálom, az pedig viszi magával a ruhadarabot. Annyiból áll, hogy mikor kattintok, akkor egy sugarat bocsátok a színtérbe (raycasting), ha pedig az elmetsz egy részecskét, akkor feljegyzem a távolságot. Ezután, ha mozgatom az egeret, miközben le van nyomva a bal gomb, újraszámolom a sugár egyenletet, viszont t paraméternek az eredeti távolságot írom be. Ennek köszönhetően, a felhasználó szemszögéből mindig a kurzort fogja követni a részecske.

Ahhoz, hogy jobban követhető legyen a szimuláció átalakítottam a színteret, mindennek egyféle színe van, a ruhadarab zöld színű, a hátsó oldala, pedig fehér,1 db fényszóró szerű fényforrás van. A ruha már a végtelen síkkal is tud ütközni.

5. Végső program

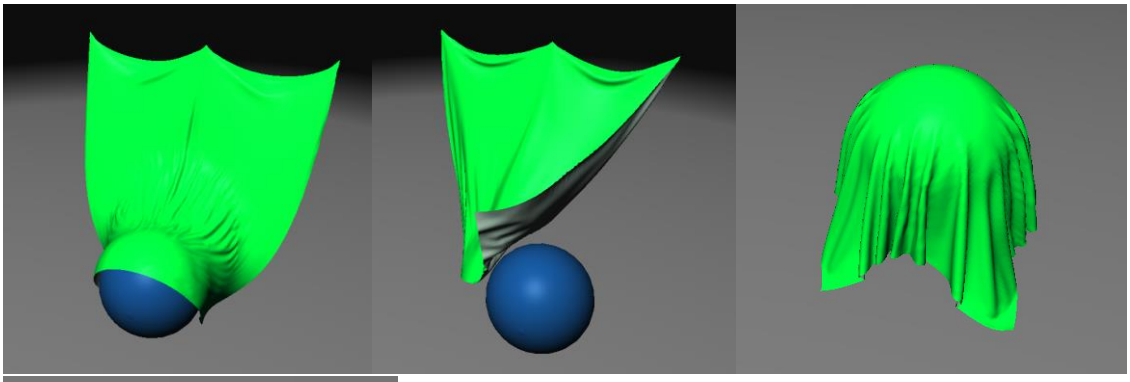
100x100-as mesh:

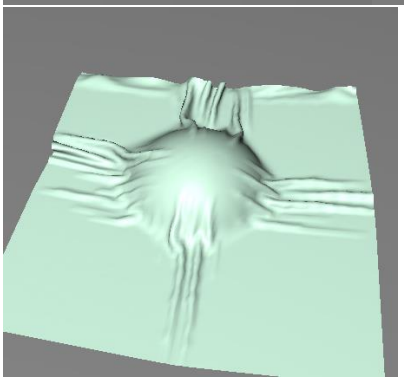
Teljesítmény:

	16 iteration	32 iteration	64 iteration
Self-collision on	55-70 FPS	50-60 FPS	40-60 FPS
Self-collision off	260-300 FPS	190-250 FPS	160-250 FPS

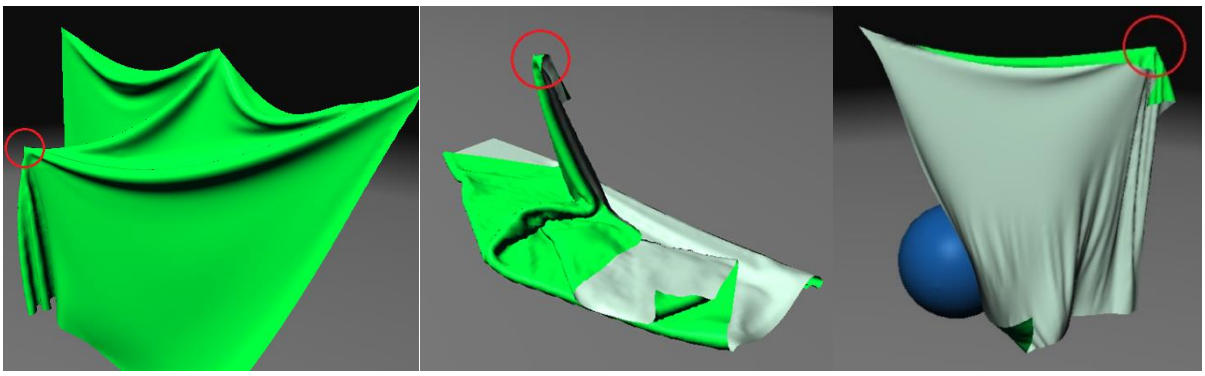
Kinézete: 32 iteráció, stiffness = 1

Ütközés a gömbbel:

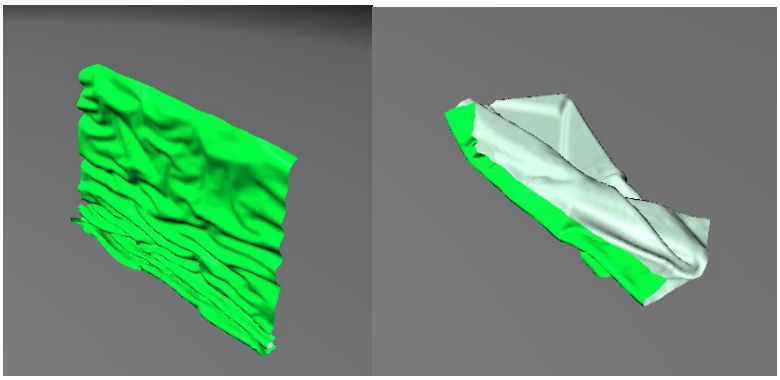




Interakció a felhasználóval:



Ütközés a talajjal és önmagával:



6. Összefoglalás

Végül a kitűzött cél sikerült, van egy interaktív demo egy ruhadarabról, ami képes ütközni több különböző módon. Nehézséget az jelentett, hogy minden új volt, ez volt az első ilyen jellegű projektem.

7. Jövőbeli munka

Tovább vinni úgy lehetne, hogy PBD világában írok egy víz szimulációt, amit összekombinálok ezzel a motorral. A végeredmény mondjuk lehetővé tenné azt, hogy vizet öntsünk egy kifeszített vászonra, ami megtartja a víz egy részét, esetleg kifolyik a szélén. Akár egy egyszerűbb modellt is ki lehet találni arra, hogy miképpen nedvesedjen a ruha.

8. Források

Matthias Müller-Fischer munkássága

- https://matthias-research.github.io/pages/publications/posBasedDyn.pdf
- https://www.youtube.com/c/matthimf
- https://www.youtube.com/c/TenMinutePhysics

Ladislav Kavan nyílt előadása

- https://www.youtube.com/watch?v=fH3VW9SaQ_c