Simulacija i vizalizacija modela tkanine

Lovrinović, Marin

Undergraduate thesis / Završni rad

2024

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

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:971182

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

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

Repository / Repozitorij:

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

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1295

SIMULACIJA I VIZALIZACIJA MODELA TKANINE

Marin Lovrinović

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1295

SIMULACIJA I VIZALIZACIJA MODELA TKANINE

Marin Lovrinović

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1295

Pristupnik: **Marin Lovrinović (0036540934)**

Studij: Elektrotehnika i informacijska tehnologija i Računarstvo

Modul: Računarstvo

Mentorica: prof. dr. sc. Željka Mihajlović

Zadatak: **Simulacija i vizalizacija modela tkanine**

Opis zadatka:

Proučiti osnove fizikalnog modela potrebnog u simulaciji tkanine. Proučiti i razraditi matematički model kojim možemo opisati tkaninu. Implementirati simulaciju modela tkanine uz pripadne komponente potrebne za vizualizaciju simulacije. Načiniti testiranje na nizu primjera. Analizirati i ocijeniti ostvarene rezultate. Diskutirati upotrebljivost ostvarenih rezultata kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Koristiti programski jezik C++ i grafičko programsko sučelje OpenGL. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 14. lipnja 2024.

Sadržaj

Uvod 1

1. Teorija 2

   1.1. Model masa i opruga 2

      1.1.1. Elastična sila opruge 2

      1.1.2. Sila prigušenja opruge 2

      1.1.3. Sila trenja 2

      1.1.4. Sila otpora zraka 3

      1.1.5. Akceleracija 3

      1.1.6. Eulerova metoda 3

      1.1.7. Kinematička jednadžba 4

      1.1.8. Verlet integracija 4

   1.2. Model čestica i ograničenja 5

   1.3. Metoda konačnih elemenata 5

2. Implementacija 6

   2.1. Razred *Cloth* 6

      2.1.1. Inicijalizacija 7

      2.1.2. Metoda *CalculateIndices* 11

      2.1.3. Metoda *CalculateNormals* 11

      2.1.4. Metoda *CalculateUVs* 13

      2.1.5. Metoda *Update* 13

   2.2. Razred *Collider* 18

   2.3. Razred *SphereCollider* 18

      2.3.1. Metoda *Displace* 19

   2.4. Razred *CubeCollider* 20

      2.4.1. Metoda *Displace* 20

2.5. Vizualizacija ... 22
3. Rezultati ... 26
3.1. Scenariji ... 26
3.2. Konstanta opruge ... 33
3.3. Dimenzije mreže ... 36
3.4. Moguća poboljšanja ... 41
Zaključak ... 42
Literatura ... 43
Sažetak ... 44
Summary ... 45
Privitak ... 46

Uvod

Simulacije u stvarnom vremenu su vrlo korisne u video igrama i drugim interaktivnim medijima. Simulacije tkanine mogu se koristiti za poboljšanje animacije ili za centralne elemente igre.

Na početku će biti istraženi različiti fizikalni modeli za reprezentaciju tkanine. Nakon toga će biti objašnjena implementacija u jeziku C++ i OpenGL-u, s naglaskom na najbitnije tehnike koje se koriste u simulaciji i vizualizaciji tkanine. Nakon toga će rezultati biti analizirani i daljnje mogućnosti razvoja projekta bit će razmotrene.

1. Teorija

Većina modela tkanine sastoji se od mreže čestica koje imaju nekakvu međusobnu interakciju.

1.1. Model masa i opruga

Ovaj model tkanine sastoji se od masa poredanih u mrežu, gdje je svaka masa povezana sa svojim susjedima elastičnim oprugama. Postoje tri vrste opruga u ovom modelu. Strukturalne koje povezuju susjede u kardinalnim smjerovima, smične koje povezuju dijagonalne susjede i pregibne koje povezuju susjede međusobno udaljene za dva mjesta [1].

1.1.1. Elastična sila opruge

Pretpostavit ćemo neku zadanu udaljenost između susjednih čestica. Ako je udaljenost veća od zadane, čestice se privlače, a ako je manja, čestice se odbijaju. Silu koja privlači dvije susjedne čestice možemo izračunati koristeći izraz (1), gdje je k konstanta opruge a dx razlika između trenutne duljine i zadane duljine.

$$F_e = k * dx \tag{1}$$

1.1.2. Sila prigušenja opruge

Sila prigušenja opruge djeluje u obrnutom smjeru od smjera kretanja čestice. Magnitudu te sile možemo izračunati koristeći izraz (2), gdje je k_p koeficijent prigušenja opruge a v brzina čestice u smjeru opruge [2].

$$F_p = k_p * v \tag{2}$$

Ta sila je bitna da zaustavimo prekomjerne vibracije tkanine i dodamo prikladan gubitak energije.

1.1.3. Sila trenja

Sila trenja može djelovati na čestice koje su u dodiru s drugim fizičkim objektima. Djeluje u suprotnom smjeru od komponente brzine čestice tangencijalne na površinu. Magnitudu

sile računamo koristeći izraz (3), gdje je k_{tr} koeficijent trenja, a F_n normalna komponenta sile kojom površina djeluje na česticu.

$$F_{tr} = k_{tr} * F_n \tag{3}$$

1.1.4. Sila otpora zraka

Sila otpora zraka djeluje na čestice koje se kreću kroz zrak, bilo to jer se kreće čestica ili se zrak kreće oko nje. Djeluje u suprotnom smjeru od smjera kretanja čestice kroz zrak. Računamo ju koristeći izraz (3), gdje je k_{tr} koeficijent otpora zraka, ρ gustoća zraka, v brzina čestice kroz zrak, a A površina čestice u smjeru kretanja kroz zrak [3].

$$F_o = \frac{1}{2} k_o * \rho * v^2 * A \tag{3}$$

1.1.5. Akceleracija

Trebamo zbrojiti sve te sile da dobijemo ukupnu silu na neku česticu. Potom možemo dobiti akceleraciju čestice koristeći izraz (4), gdje je m masa čestice, a F_{uk} ukupna sila na česticu. Ukupnoj akceleraciji možemo dodati i akceleraciju zbog gravitacije.

$$a = \frac{F_{uk}}{m} \tag{4}$$

Sada kada znamo akceleraciju čestice, postoji nekoliko načina za numeričku integraciju brzine i pozicije čestice.

1.1.6. Eulerova metoda

Najjednostavnija metoda numeričke integracije je Eulerova metoda. Pamtimo poziciju i brzinu čestice. Za svaki vremenski korak izračunamo novu poziciju čestice koristeći izraz (5).

$$x(t + \Delta t) = x(t) + v(t) * \Delta t \tag{5}$$

Nakon toga izračunamo novu brzinu čestice koristeći izraz (6).

$$v(t + \Delta t) = v(t) + a(t) * \Delta t \tag{6}$$

Ova metoda je vrlo brza, ali nije vrlo precizna. Ne uračunava činjenicu da akceleracija djeluje kroz cijeli vremenski korak, a ne samo na kraju.

1.1.7. Kinematička jednadžba

Pamtimo poziciju i brzinu čestice. Ova metoda pretpostavlja konstantnu akceleraciju unutar vremenskog koraka i uračunava da se brzina konstantno mijenja. Izračunavamo novu poziciju pomoću izraza (7).

$$x(t + \Delta t) = x(t) + v(t) * \Delta t + \frac{a(t) * \Delta t^2}{2} \tag{7}$$

Nakon toga na isti način kao u Eulerovoj metodi izračunamo novu brzinu čestice koristeći izraz (8).

$$v(t + \Delta t) = v(t) + a(t) * \Delta t \tag{8}$$

Metoda je preciznija, ali sporija od Eulerove.

1.1.8. Verlet integracija

Zamislimo da želimo izračunati prethodnu poziciju čestice. Taj izračun bi izgledao kao izraz (7).

$$x(t - \Delta t) = x(t) - v(t) * \Delta t + \frac{a(t) * \Delta t^2}{2} \tag{9}$$

Ako sada zbrojimo izraze (7) i (9), dobiti ćemo izraz (10).

$$x(t + \Delta t) + x(t - \Delta t) = 2x(t) + a(t) * \Delta t^2 \tag{10}$$

Ako izrazimo poziciju u sljedećem vremenskom koraku, dobiti ćemo izraz (11).

$$x(t + \Delta t) = 2x(t) - x(t - \Delta t) + a(t) * \Delta t^2 \tag{11}$$

Iz toga vidimo da za ovu metodu ne trebamo pamtiti brzinu čestice, već samo trenutnu i prethodnu poziciju. Ova metoda je slične brzine Eulerovoj i vrlo je precizna [4].

1.2. Model čestica i ograničenja

Ovaj model također dijeli tkaninu na mrežu čestica, ali ne modelira veze između susjednih čestica s oprugama i silama, već direktno ograničava međusobne pozicije čestica. Uglavnom to znači da susjedne čestice pokušavaju ostati na zadanoj udaljenosti. Tim ograničenjima se mogu dodati dodatna svojstva poput rastezljivosti i savitljivosti. Verlet integracija je vrlo korisna u ovakvom modelu jer dobro reagira na direktno mijenjanje pozicije čestice.

1.3. Metoda konačnih elemenata

Metoda konačnih elemenata dijeli tkaninu na mrežu manjih, jednostavnijih elemenata (obično trokuta ili četverokuta). Svaki element se ponaša prema zakonima fizike, a ukupno ponašanje tkanine dobiva se rješavanjem jednadžbi koje upravljaju interakcijama između tih elemenata. Metoda konačnih elemenata je vrlo precizna, ali uglavnom previše računalno zahtjevna za primjene u stvarnom vremenu.

2. Implementacija

Model korišten u implementaciji simulacije tkanine je model masa i opruga. Korištena metoda integracije je Verlet integracija. Pretpostavlja se korištenje osnovnog grafičkog pogona u OpenGL-u. Omogućeno je micanje kamere s mišem i tipkovnicom za vrijeme izvršavanja simulacije.

2.1. Razred *Cloth*

Razred Cloth je najbitni razred u ovoj implementaciji simulacije tkanine:

```
class Cloth {
private:
    int dim;
    vector<vector<glm::vec3>> previousPositions;
    vector<vector<glm::vec3>> currentPositions;
    vector<vector<glm::vec3>> nextPositions;
    vector<vector<glm::vec3>> currentNormals;
    vector<vector<glm::vec3>> currentVelocities;
    vector<vector<glm::vec4>> colliderSurfaces;
    vector<vector<bool>> fixed;
    float springConstant;
    float springDampingCoefficient;
    float dragCoefficient;

    float equilibriumDistance;
    vector<glm::ivec2> neighborOffsets;
    map<int, map<int, float>> neighborDistances;

    TriangleMesh* mesh;
public:
    Object* object;

    Cloth(int dimension, float springConstant, float
springDampingCoefficient, float dragCoefficient, Shader*
shader,
```

```
          glm::vec3 position, float sideLength, bool
vertical, vector<glm::ivec2> fixed);
     void Update(float dt, glm::vec3 gravity, glm::vec3
airflow, const vector<Collider*>& colliders);
};
```

Kod 2.1 Razred *Cloth*

Korisnik razreda može specificirati dimenziju mreže (broj točaka u mreži, na primjer 20 x 20), konstantu opruga u mreži, koeficijent prigušenja opruga, koeficijent otpora zraka, poziciju tkanine, duljinu tkanine, orijentaciju (vertikalna ili vodoravna) i listu nepomičnih točaka u tkanini. Potrebna su nam tri dvodimenzionalna spremnika (buffer-a) s pozicijama točaka u mreži. Spremnik `currentPositions` drži trenutnu poziciju točaka, `previousPositions` drži prethodnu poziciju točaka koja nam je potrebna za Verlet integraciju, a `nextPositions` drži sljedeću poziciju točaka. Naime, `nextPositions` je isključivo mjesto gdje ćemo zapisati sljedeće izračunate pozicije točaka, pa nakon toga zamijeniti `nextPositions` i `currentPositions`. Ovo je klasični pristup s dvostrukim spremnikom (engl. double buffering). Ne smijemo zapisivati nove pozicije direktno u `currentPositions` zato što sile koje djeluju na točku ovise o pozicijama susjeda te točke. Ako bi mijenjali te pozicije prije nego izračunamo silu na svaku točku, onda bi rezultat simulacije ovisio o redoslijedu obrađivanja točaka i simulacija ne bi bila precizna. Ako bismo paralelizirali ovaj postupak, onda simulacija ne bi bila niti deterministička, jer bi istovremeno čitali i pisali iz iste memorije. Pristup s dvostrukim spremnikom nam pomaže da izbjegnemo taj problem. Normale tkanine spremamo u `currentNormals` jer su nam potrebne za računanje otpora zraka. Unatoč tome što koristimo Verlet integraciju, spremamo brzine točaka u mreži `currentVelocities` zbog računanja prigušenja u oprugama. U `colliderSurfaces` spremamo podatke o površinama sudarača koje pojedine točke u mreži dodiruju. Članovi `mesh` i `object` služe za vizualizaciju tkanine u grafičkom pogonu.

2.1.1. Inicijalizacija

Ovako se inicijaliziraju podatci o tkanini:

```
Cloth::Cloth(int dimension, float springConstant, float
springDampingCoefficient, float dragCoefficient, Shader*
shader,
             glm::vec3 position, float sideLength, bool
vertical, vector<glm::ivec2> fixed) :
dim(dimension),
springConstant(springConstant),
springDampingCoefficient(springDampingCoefficient),
dragCoefficient(dragCoefficient) {
    previousPositions = vector<vector<glm::vec3>>(dim,
vector<glm::vec3>(dim));
    currentPositions = vector<vector<glm::vec3>>(dim,
vector<glm::vec3>(dim));
    nextPositions = vector<vector<glm::vec3>>(dim,
vector<glm::vec3>(dim));
    currentNormals = vector<vector<glm::vec3>>(dim,
vector<glm::vec3>(dim));
    currentVelocities = vector<vector<glm::vec3>>(dim,
vector<glm::vec3>(dim));
    colliderSurfaces = vector<vector<glm::vec4>>(dim,
vector<glm::vec4>(dim));
    this->fixed = vector<vector<bool>>(dim,
vector<bool>(dim));

    equilibriumDistance = sideLength / (dimension - 1);
    neighborOffsets = {
            {-2, -1},
            {-2, 0},
            {-2, 1},
            {-1, -2},
            {-1, -1},
            {-1, 0},
            {-1, 1},
            {-1, 2},
            {0, -2},
            {0, -1},
            {0, 1},
            {0, 2},
            {1, -2},
            {1, -1},
            {1, 0},
```

```
            {1, 1},
            {1, 2},
            {2, -1},
            {2, 0},
            {2, 1}
    };
    for (glm::ivec2 n : neighborOffsets) {
        neighborDistances[n.x][n.y] = sqrt(n.x * n.x + n.y *
n.y);
    }

    glm::vec3 corner;
    if (vertical) {
        corner = position + glm::vec3(-0.5f * sideLength, -
0.5f * sideLength, 0);
    } else {
        corner = position + glm::vec3(-0.5f * sideLength, 0,
-0.5f * sideLength);
    }
    for (int i = 0; i < dim; ++i) {
        for (int j = 0; j < dim; ++j) {
            glm::vec3 particlePos;
            if (vertical) {
                particlePos = corner + glm::vec3(i *
equilibriumDistance, j * equilibriumDistance, 0);
            } else {
                particlePos = corner + glm::vec3(i *
equilibriumDistance, 0, j * equilibriumDistance);
            }
            currentPositions[i][j] = particlePos;
            previousPositions[i][j] = particlePos;
            currentVelocities[i][j] = glm::vec3(0);
            colliderSurfaces[i][j] = glm::vec4(0);
            this->fixed[i][j] = false;
        }
    }
    for (glm::ivec2 f : fixed) {
        this->fixed[f.x][f.y] = true;
    }

    vector<glm::vec3> meshVertices(dim * dim);
```

```
    TwoDimToOneDim(currentPositions, meshVertices);

    vector<int> meshIndices =
CalculateIndices(currentPositions.size());

    CalculateNormals(currentPositions, currentNormals);
    vector<glm::vec3> meshNormals(dim * dim);
    TwoDimToOneDim(currentNormals, meshNormals);

    vector<glm::vec3> uvs =
CalculateUVs(currentPositions.size());

    mesh = new TriangleMesh(meshVertices, meshNormals,
meshIndices, uvs);
    mesh->SendToGpu();
    object = new Object(mesh, shader);
}
```

Kod 2.1.1 Konstruktor razreda *Cloth*

Izračunata je udaljenost najbližih susjeda u ravnotežnom položaju `equilibriumDistance`. Inicijaliziran je vektor `neighborOffset` koji sadrži pomake svih susjeda koji će utjecati na pojedinu točku u mreži. Razmatramo 20 najbližih susjeda (isto kao pikselizirani krug dimenzija 5x5). To uključuje strukturalne, smične i pregibne veze, uz neke dodatne. Izračunate su relativne udaljenosti susjeda ovisno o pomaku zato da taj posao ne moramo ponavljati kasnije.

Pozicije čestica su inicijalizirane prema zadanim parametrima i iz tih pozicija su izračunati podatci o mreži poligona grafičkog objekta. Funkcija `TwoDimToOneDim` služi za pretvaranje 2D vektora u 1D, pošto nam je potreban kontinuirani dio memorije koji ćemo slati na grafičku karticu. Pozicije točaka u modelu tkanine postaju pozicije vrhova u mreži poligona. Funkcija `CalculateIndices` izračunava indekse, `CalculateNormals` izračunava normale vrhova i `CalculateUVs` izračunava UV koordinate vrhova. Metoda `SendToGpu` šalje podatke o mreži poligona na grafičku karticu koristeći uobičajeni postupak u OpenGL-u.

2.1.2. Metoda *CalculateIndices*

```
vector<int> CalculateIndices(int dim){
    vector<int> indices;
    for (int i = 0; i < dim - 1; ++i) {
        for (int j = 0; j < dim - 1; ++j) {
            indices.emplace_back(i * dim + j);
            indices.emplace_back(i * dim + (j + 1));
            indices.emplace_back((i + 1) * dim + j);

            indices.emplace_back(i * dim + (j + 1));
            indices.emplace_back((i + 1) * dim + (j + 1));
            indices.emplace_back((i + 1) * dim + j);
        }
    }
    return indices;
}
```

Kod 2.1.2 Metoda *CalculateIndices*

Indeksi se računaju tako da prođemo kroz svaki kvadrat koji tvore točke mreže tkanine i „popločamo“ ga sa dva trokuta. Dvostrano renderiranje je omogućeno u grafičkom pogonu.

2.1.3. Metoda *CalculateNormals*

```
void CalculateNormals(const vector<vector<glm::vec3>>&
currentPositions, vector<vector<glm::vec3>>& normals) {
    int dim = currentPositions.size();
    for (int i = 0; i < dim - 1; ++i) {
        for (int j = 0; j < dim - 1; ++j) {
            normals[i][j] = glm::vec3(0);
        }
    }

    for (int i = 0; i < dim - 1; ++i) {
        for (int j = 0; j < dim - 1; ++j) {

            glm::vec3 faceNormal1 =
```

```
glm::normalize(glm::cross(currentPositions[i][j + 1] -
currentPositions[i][j],

currentPositions[i + 1][j] - currentPositions[i][j]));
            normals[i][j] += faceNormal1;
            normals[i][j + 1] += faceNormal1;
            normals[i + 1][j] += faceNormal1;

            glm::vec3 faceNormal2 =

glm::normalize(glm::cross(currentPositions[i + 1][j] -
currentPositions[i + 1][j + 1],

currentPositions[i][j + 1] - currentPositions[i + 1][j +
1]));
            normals[i][j + 1] += faceNormal2;
            normals[i + 1][j + 1] += faceNormal2;
            normals[i + 1][j] += faceNormal2;
        }
    }

    for (int i = 0; i < dim - 1; ++i) {
        for (int j = 0; j < dim - 1; ++j) {
            normals[i][j] = glm::normalize(normals[i][j]);
        }
    }
}
```

Kod 2.1.3 Metoda *CalculateNormals*

Normale vrhova se računaju tako da izračunamo prosjek normala svih susjednih trokuta za svaki vrh. To ćemo postići tako da prvo vrijednost normala za sve vrhove inicijaliziramo na nul-vektor. Nakon toga za svaki trokut izračunamo normalu i zbrojimo tu normalu s dosadašnjim vrijednostima normala od sva tri vrha tog trokuta. Na kraju normaliziramo sve vrijednosti normala vrhova trokuta [5].

2.1.4. Metoda *CalculateUVs*

```
vector<glm::vec3> CalculateUVs(int dim) {
    vector<glm::vec3> uvs;
    for (int i = 0; i < dim; ++i) {
        for (int j = 0; j < dim; ++j) {
            uvs.emplace_back((float)i / (dim - 1), (float)j /
(dim - 1), 0);
        }
    }
    return uvs;
}
```

Kod 2.1.4 Metoda *CalculateUVs*

UV koordinate se računaju tako da interpoliramo koordinate od 0 do 1 ovisno o indeksima točke.

2.1.5. Metoda *Update*

Metoda Update se poziva svaki frame i služi za izračunavanje novih pozicija svih točaka u mreži:

```
void Cloth::Update(float dt, glm::vec3 gravity, glm::vec3
airflow, const vector<Collider*>& colliders) {
    for (int i = 0; i < dim; ++i) {
        for (int j = 0; j < dim; ++j) {
            if (fixed[i][j]) {
                nextPositions[i][j] = currentPositions[i][j];
                continue;
            }

            glm::vec3 particlePosition =
currentPositions[i][j];
            glm::vec3 particleVelocity =
currentVelocities[i][j];
            glm::vec3 acceleration = gravity;
```

```
            for (glm::ivec2 neighborOffset : neighborOffsets)
{
                int neighborI = i + neighborOffset.x;
                int neighborJ = j + neighborOffset.y;
                if (neighborI < 0 || dim <= neighborI ||
neighborJ < 0 || dim <= neighborJ) {
                    continue;
                }

                glm::vec3 neighborPosition =
currentPositions[neighborI][neighborJ];
                float unitDistancesFromParticle =
neighborDistances[neighborOffset.x][neighborOffset.y];
                float neighborEquilibriumDistance =
equilibriumDistance * unitDistancesFromParticle;

                float neighborDistance =
glm::distance(particlePosition, neighborPosition);
                glm::vec3 directionToNeighbor =
(neighborPosition - particlePosition) / neighborDistance;
                float dx = neighborDistance -
neighborEquilibriumDistance;
                glm::vec3 springAcceleration =
directionToNeighbor * dx * springConstant;

                glm::vec3 neighborVelocity =
currentVelocities[neighborI][neighborJ];
                glm::vec3 relativeVelocity = particleVelocity
- neighborVelocity;
                glm::vec3 springDampingAcceleration = -
springDampingCoefficient
                        * glm::dot(relativeVelocity,
directionToNeighbor) * directionToNeighbor;

                acceleration += (springAcceleration +
springDampingAcceleration) / unitDistancesFromParticle;
            }

            // calculate air resistance
```

```
            glm::vec3 velocityInAir = particleVelocity -
airflow;

            acceleration += -dragCoefficient
                    * abs(glm::dot(velocityInAir,
currentNormals[i][j])) * velocityInAir;

            // calculate friction
            glm::vec4 colliderSurface =
colliderSurfaces[i][j];
            if (colliderSurface != glm::vec4(0)) {
                glm::vec3 colliderNormal = colliderSurface;
                float frictionCoefficient =
colliderSurface.w;

                glm::vec3 tangentialVelocity =
particleVelocity
                        - glm::dot(particleVelocity,
colliderNormal) * colliderNormal;

                float tangentialSpeed =
glm::length(tangentialVelocity);
                if (tangentialSpeed > 0) {
                    glm::vec3 tangentialVelocityDirection =
tangentialVelocity / tangentialSpeed;
                    float surfaceNormalForce = max(0,
glm::dot(-acceleration, colliderNormal));
                    acceleration += -frictionCoefficient *
tangentialVelocityDirection * surfaceNormalForce;
                }
            }

            glm::vec3 nextPosition = 2.0f * particlePosition
- previousPositions[i][j] + acceleration * dt * dt;

            // calculate collisions
            colliderSurfaces[i][j] = glm::vec4(0);
            for (Collider *const collider : colliders) {
                colliderSurfaces[i][j] = collider-
>Displace(particlePosition, nextPosition);
            }
```

```
                nextPositions[i][j] = nextPosition;
            }
        }
        std::swap(previousPositions, currentPositions);
        std::swap(currentPositions, nextPositions);

        for (int i = 0; i < dim; ++i) {
            for (int j = 0; j < dim; ++j) {
                currentVelocities[i][j] = (currentPositions[i][j]
    - previousPositions[i][j]) / dt;
            }
        }

        TwoDimToOneDim(currentPositions, mesh->vertices);

        CalculateNormals(currentPositions, currentNormals);
        TwoDimToOneDim(currentNormals, mesh->normals);

        mesh->SendToGpu();
    }
```

Kod 2.1.5 Metoda *Update*

Ova funkcija sadrži najbitniji dio simulacije tkanine. Prolazimo kroz sve točke u tkanini i računamo nove pozicije. Za svaku točku trebamo napraviti sljedeći postupak.

Prvo trebamo izračunati ukupnu akceleraciju na točku. Pretpostavljena masa točke je jedan, tako da možemo sile odmah tretirati kao akceleraciju i zbrajati ih u ukupnu akceleraciju na točku. Postoji nekoliko izvora sile (akceleracije) na točku. Osnovna akceleracija je prema dolje zbog gravitacije.

Susjedne točke su oprugama povezane s našom točkom. Prolazimo kroz 20 najbližih susjeda (one koji postoje). Za susjede koji su dalje moramo prilagoditi `equilibriumDistance`. Već smo izračunali koliko jediničnih udaljenosti je svaki susjed udaljen (`neighborDistances`), pa sada množimo `equilibriumDistance` s tim brojem. Također prilagođavamo ukupnu silu kojom susjed djeluje na točku tako da ju podijelimo s brojem jediničnih udaljenosti od naše točke. Dalji susjedi slabije djeluju na točku. Smjer vektora akceleracije je pravac koji prolazi kroz našu točku i susjeda, a duljinu i orijentaciju vektora određuju jednadžba opruge i jednadžba prigušenja opruge.

Nakon što smo izračunali akceleraciju zbog susjednih opruga, dodajemo otpor zraka. Dobivamo brzinu čestice u zraku tako da oduzmemo brzinu kretanja zraka od brzine naše čestice. Smjer akceleracije je suprotan smjeru kretanja čestice kroz zrak, a proporcionalan je kvadratu brzine. Ako se čestica kreće u smjeru normale tkanine, onda hvata više zraka i otpor je veći. Ako se kreće okomito na normalu tkanine, hvata manje zraka i otpor je manji. Na taj način smo modelirali površinu iz jednadžbe otpora zraka. To nam omogućuje da dobijemo „lelujanje“ tkanine na vjetru.

Sljedeća sila koja djeluje na našu točku je uzrokovana potencijalnim trenjem sa sudaračem (engl. collider). Smjer trenja je suprotan od komponente brzine tangencijalne na površinu sudarača. Magnitudu sile trenja određuje jednadžba trenja. Da bismo dobili normalnu silu površine, pretpostavljamo da sudarač djeluje na točku suprotnom silom od svih ostalih sila. Nalazimo komponentu te sile koja je normalna na površinu sudarača da bi dobili trenje.

Nakon što izračunamo ukupnu akceleraciju na točku, izračunamo njenu sljedeću poziciju koristeći Verlet integraciju. Nakon toga moramo osigurati da je ta pozicija izvan svih sudarača. Postoje sudarači u obliku kugle i kocke. Popravak nove pozicije točke i računanje normale površine na mjestu sudara je odgovornost sudarača i objasnit ćemo ju poslije. Ove izračunate normale sudarača ćemo koristiti sljedeći put kada se pozove metoda `Update` da izračunamo trenje na česticu. Točke koje ne dodiruju niti jedan sudarač imat će nul-vektor kao spremljenu normalu površine.

Nakon što smo osigurali da je pozicija izvan sudarača, tu sljedeću poziciju ćemo zapisati u spremnik `nextPosition`. Kada to napravimo za sve točke, zamijenit ćemo spremnike `previousPosition` i `currentPosition`, pa nakon toga `currentPosition` i `nextPosition`. Na taj način postižemo kružni pomak spremnika. Vrijednosti iz `previousPosition` nam više ne trebaju pa te vrijednosti odlaze u `nextPosition` koji ćemo prepisati sljedeći put kada bude pozvana metoda `Update`. Vrijednosti iz `currentPosition` odlaze u `previousPosition` i koristit ćemo ih za Verlet integraciju. Vrijednosti iz `nextPosition` odlaze u `currentPosition` i postaju nove aktualne pozicije točaka.

Nakon toga izračunamo nove brzine točaka s obzirom na nove pozicije. Napokon iz novih pozicija točaka računamo nove pozicije vrhova i nove normale mreže poligona objekta, pa ih šaljemo na grafičku karticu. Ne trebamo ponovno računati indekse niti UV koordinate jer se osnovni odnosi između vrhova i trokuta u mreži nisu promijenili.

2.2. Razred *Collider*

Razred `Collider` je apstraktni razred koji nam je potreban da bi u metodu `Update` mogli slati sudarač u obliku kugle ili u obliku kocke, a da ih metoda `Update` ne mora znati razlikovati. To nam omogućuje da dodamo još vrsta sudarača u budućnosti.

```
class Collider {
public:
    virtual glm::vec4 Displace(glm::vec3 particlePosition,
glm::vec3& particleNextPosition) const = 0;
    virtual ~Collider() = default;
};
```

Kod 2.2 Razred *Collider*

Odgovornost metode `Displace` je mijenjanje `particleNextPosition` tako da bude izvan sudarača i vraćanje normale površine na mjestu sudara kao prve tri komponente vektora i koeficijent trenja kao četvrtu komponentu vektora vraćenog iz metode.

2.3. Razred *SphereCollider*

Razred `SphereCollider` predstavlja sudarač u obliku kugle. Nasljeđuje razred `Collider`.

```
class SphereCollider : public Collider {
public:
    glm::vec3 position;
    float radius;
    float frictionCoefficient;
    SphereCollider(glm::vec3 position, float radius, float
frictionCoefficient);
    glm::vec4 Displace(glm::vec3 particlePosition, glm::vec3&
particleNextPosition) const override;
```

```
};
```

Kod 2.3 Razred *SphereCollider*

Sadrži poziciju (centar kugle), radijus kugle i koeficijent trenja.

2.3.1. Metoda *Displace*

Implementacija metode `Displace` za sudarače u obliku kugle:

```
glm::vec4 SphereCollider::Displace(glm::vec3
particlePosition, glm::vec3 &particleNextPosition) const {
    float distanceFromCenter =
glm::distance(particleNextPosition, position);
    if (distanceFromCenter >= radius) {
        return glm::vec4(0);
    }

    glm::vec3 normal = (particleNextPosition - position) /
distanceFromCenter;

    particleNextPosition = position + normal * radius;

    return { normal, frictionCoefficient };
}
```

Kod 2.2.1 Metoda *SphereCollider::Displace*

Metoda provjerava je li udaljenost nove pozicije točke veća od radijusa kugle. Ako je, točka je izvan kugle i vraćamo nul-vektor. Međutim, ako je unutar kugle, onda jednostavno nalazimo najbližu poziciju na površini kugle i tamo premještamo novu poziciju točke. U ovom slučaju ne koristimo `particlePosition`, već samo `particleNextPosition`.

2.4. Razred *CubeCollider*

Razred `CubeCollider` predstavlja sudarač u obliku kocke. Nasljeđuje razred `Collider`.

```
class CubeCollider : public Collider {
public:
    glm::vec3 position;
    float sideLength;
    float frictionCoefficient;
    CubeCollider(glm::vec3 position, float sideLength, float
frictionCoefficient);
    glm::vec4 Displace(glm::vec3 particlePosition, glm::vec3&
particleNextPosition) const override;
};
```

Kod 2.4 Razred *CubeCollider*

Sadrži poziciju (centar kocke), duljinu stranice kocke i koeficijent trenja.

2.4.1. Metoda *Displace*

Implementacija metode `Displace` za sudarače u obliku kocke:

```
glm::vec4 CubeCollider::Displace(glm::vec3 particlePosition,
glm::vec3 &particleNextPosition) const {
    float halfSide = sideLength / 2;

    glm::vec3 fromCenter = particleNextPosition - position;
    if (abs(fromCenter.x) >= halfSide || abs(fromCenter.y) >=
halfSide || abs(fromCenter.z) >= halfSide) {
        return glm::vec4(0);
    }

    // raycast against 3 planes from inside of cube
```

```
    glm::vec3 pos = particleNextPosition;
    glm::vec3 dir = particlePosition - particleNextPosition;
    glm::vec3 dirSign = glm::vec3(copysign(1, dir.x),
copysign(1, dir.y), copysign(1, dir.z));

    glm::vec3 normal;
    float t = std::numeric_limits<float>::infinity();

    if (dir.x != 0) {
        float faceX = position.x + halfSide * dirSign.x;
        float tx = (faceX - pos.x) / dir.x;
        if (tx < t) {
            t = tx;
            normal = glm::vec3(dirSign.x, 0, 0);
        }
    }

    if (dir.y != 0) {
        float faceY = position.y + halfSide * dirSign.y;
        float ty = (faceY - pos.y) / dir.y;
        if (ty < t) {
            t = ty;
            normal = glm::vec3(0, dirSign.y, 0);
        }
    }

    if (dir.z != 0) {
        float faceZ = position.z + halfSide * dirSign.z;
        float tz = (faceZ - pos.z) / dir.z;
        if (tz < t) {
            t = tz;
            normal = glm::vec3(0, 0, dirSign.z);
        }
    }

    if (t == std::numeric_limits<float>::infinity()) {
        return glm::vec4(0);
    }

    particleNextPosition = pos + dir * t;
```

```
    return { normal, frictionCoefficient };
}
```

Kod 2.4.1 Metoda *CubeCollider::Displace*

Prvo provjeravamo je li udaljenost od centra kocke po bilo kojoj od dimenzija veća od pola stranice kocke. Ako je, točka je izvan kocke. Međutim, ako je unutar kocke, tada bacamo zraku od `particleNextPosition` u smjeru `particlePosition` (iz unutrašnjosti kocke prema van). Pokušavamo pogoditi sve tri ravnine u kojima leže tri stranice kocke u smjeru bacanja zrake. Ravnina koju prvu pogodimo (najbliža) je ona kroz koju je prošla točka (uz pretpostavku pravocrtnog kretanja između zadnje dvije pozicije). Poziciju koju smo tada pogodili je pozicija kroz koju je točka ušla u kocku i tamo ćemo pomaknuti novu poziciju.

Jednostavniji postupak (pomicanje točke na najbližu poziciju na površini kocke) nije korišten jer nije bio dovoljno stabilan blizu rubova kocke. Ovaj precizniji postupak je stabilan.

2.5. Vizualizacija

Implementacija vizualizacije je bila jednostavna s obzirom da smo već izračunali podatke o mreži 3D objekta tkanine. Ovo je sjenčar vrhova koji koristimo za vizualizaciju tkanine (i sudarača):

```
#version 330 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec3 aUV;

out vec3 normal;
out vec3 position;
out vec3 uv;

uniform mat4 model;
```

```
uniform mat4 viewProjection;

void main() {
    mat3 normalTransformation =
mat3(transpose(inverse(model)));
    normal = normalize(normalTransformation * aNormal);
    position = vec3(model * vec4(aPos, 1.0));
    uv = aUV;
    gl_Position = viewProjection * vec4(position, 1);
}
```

Kod 2.5.1 Sjenčar vrhova

Koristimo klasični pristup sa matricama modela, pogleda i projekcije. U slučaju tkanine matrica modela ne radi puno s obzirom da smo direktno mijenjali podatke o mreži, a transformacija samog objekta je ostala nepromijenjena.

Ovo je sjenčar fragmenata:

```
#version 330 core

in vec3 normal;
in vec3 position;
in vec3 uv;

uniform vec3 eye;

uniform vec3 lightPosition;
uniform vec3 lightAmbient;
uniform vec3 lightDiffuse;
uniform vec3 lightSpecular;

uniform vec3 materialAmbient;
uniform vec3 materialDiffuse;
uniform vec3 materialSpecular;
uniform float materialShininess;

uniform vec3 backgroundColor;
```

```
uniform vec3 curveColor;
uniform float curvature;
uniform float curveThickness;
uniform float curveFrequency;

out vec4 FragColor;

void main() {
    vec3 normalizedNormal = normalize(normal);
    normalizedNormal = gl_FrontFacing ? normalizedNormal : -
normalizedNormal;

    vec3 toEye = normalize(eye - position);
    vec3 toLight = lightPosition - position;
    float distanceToLight = length(toLight);
    toLight /= distanceToLight;

    vec3 ambient = materialAmbient * lightAmbient;
    vec3 diffuse = materialDiffuse *
clamp(dot(normalizedNormal, toLight), 0, 1) * lightDiffuse;

    float specularMultiplier = pow(max(0, dot(toEye,
reflect(-toLight, normalizedNormal))), materialShininess);
    vec3 specular = materialSpecular * specularMultiplier *
lightSpecular;

    vec3 light = ambient + (diffuse + specular) /
(distanceToLight + 0.00001);
    float curve = clamp(sin(((uv.x + uv.y) * 50 + curvature *
sin((uv.x - uv.y) * 50 * curveFrequency)) / curveThickness),
0, 1);

    light *= mix(backgroundColor, curveColor, curve);

    FragColor = vec4(light, 1.0);
}
```

Kod 2.5.1 Sjenčar fragmenata

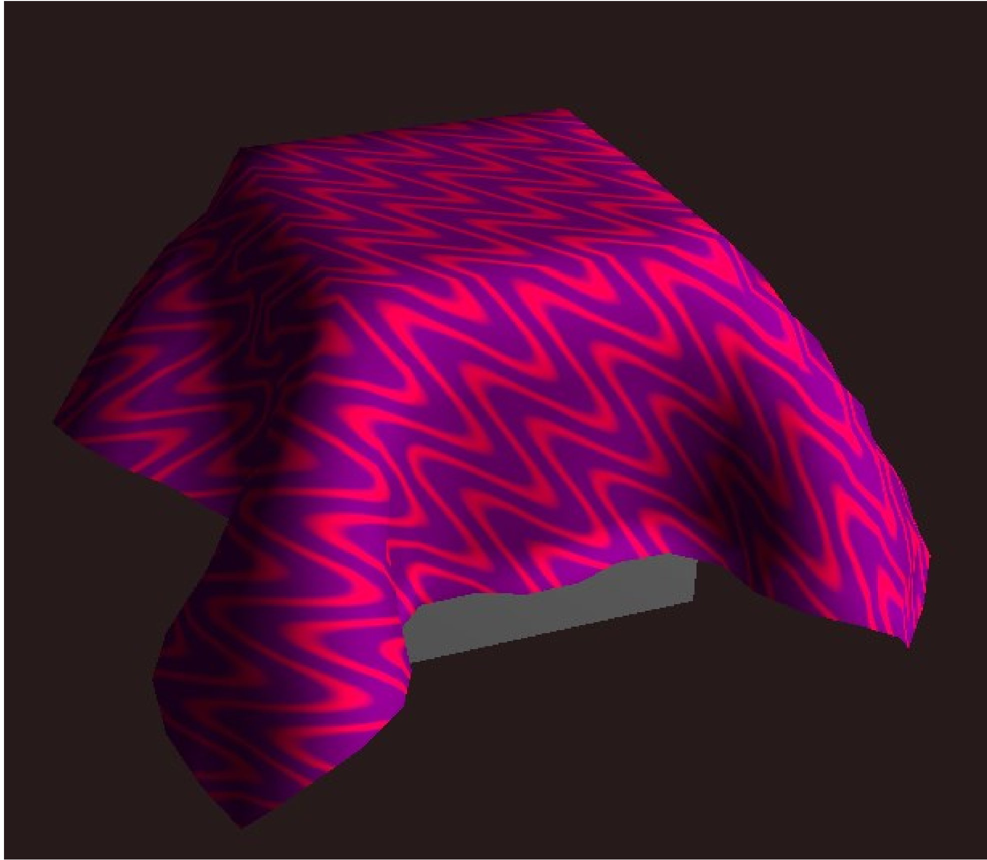
Ovo je standardni Phongov model osvjetljenja s ambijentalnom, difuznom i zrcalnom komponentom osvjetljenja. Dodana je mogućnost crtanja krivulja na površini objekta radi boljeg razumijevanja i jasnije vizualizacije savijanja tkanine.

3. Rezultati

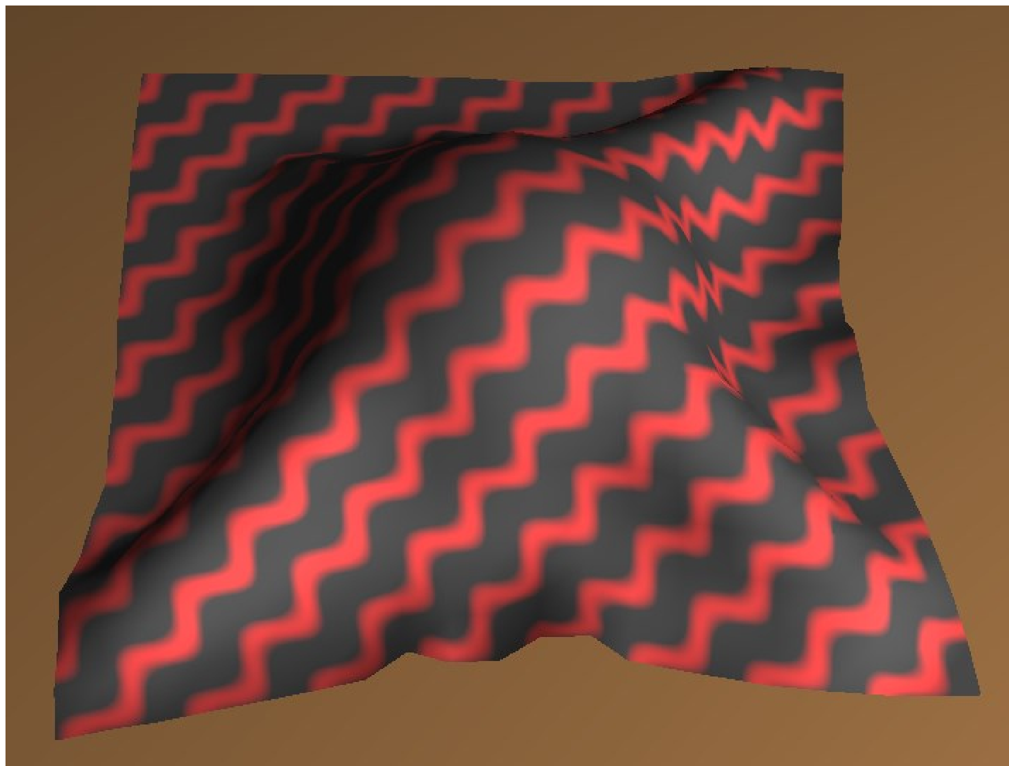
3.1. Scenariji

Promotrimo kako se tkanina ponaša u različitim scenarijima s obzirom na poziciju, ograničenja, sudarače i vjetar.

Slika 3.1.1 – Tkanina na kugli

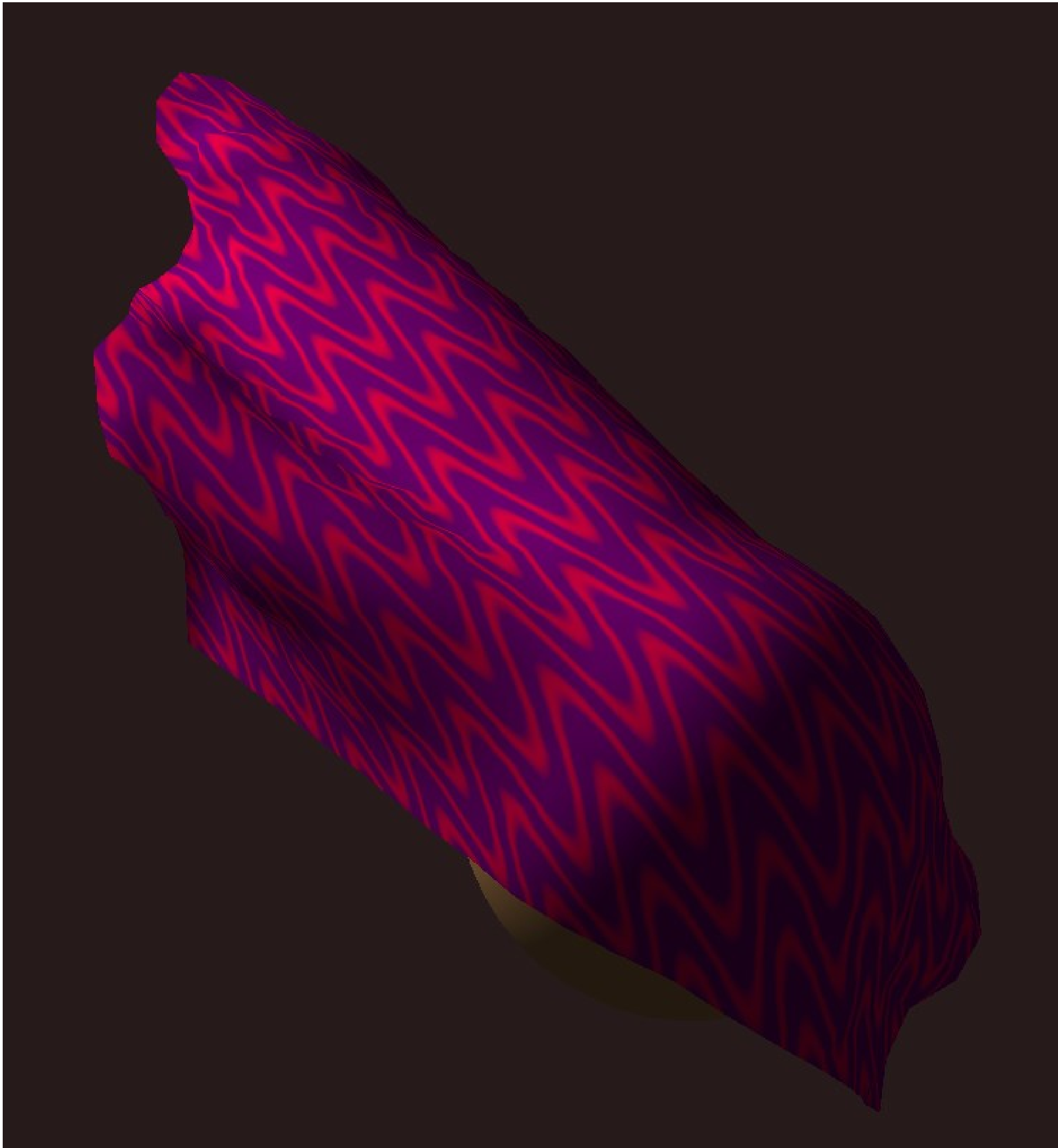


Slika 3.1.2 – Tkanina na kocki



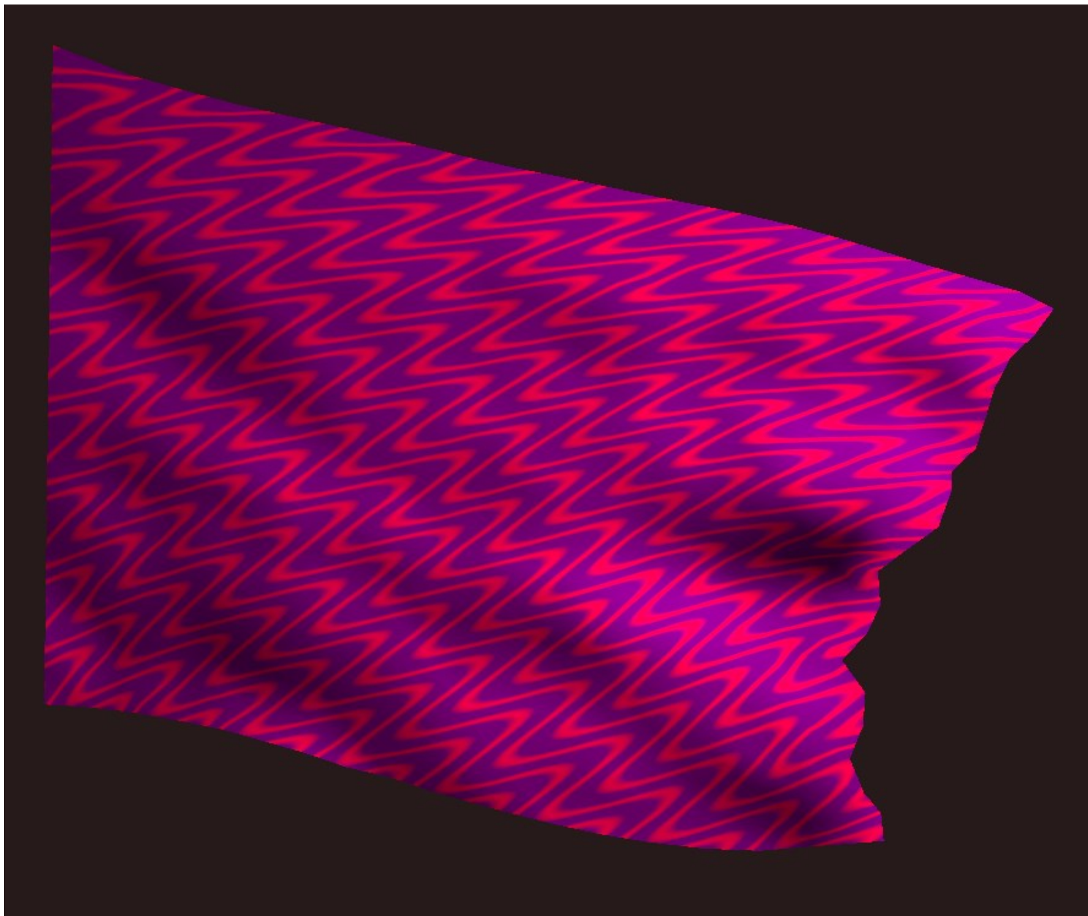
Slika 3.1.3 – Tkanina na kugli koja je na podu

Obje vrste sudarača dobro funkcioniraju. Na slici 3.1.3 je pod samo velika kocka.

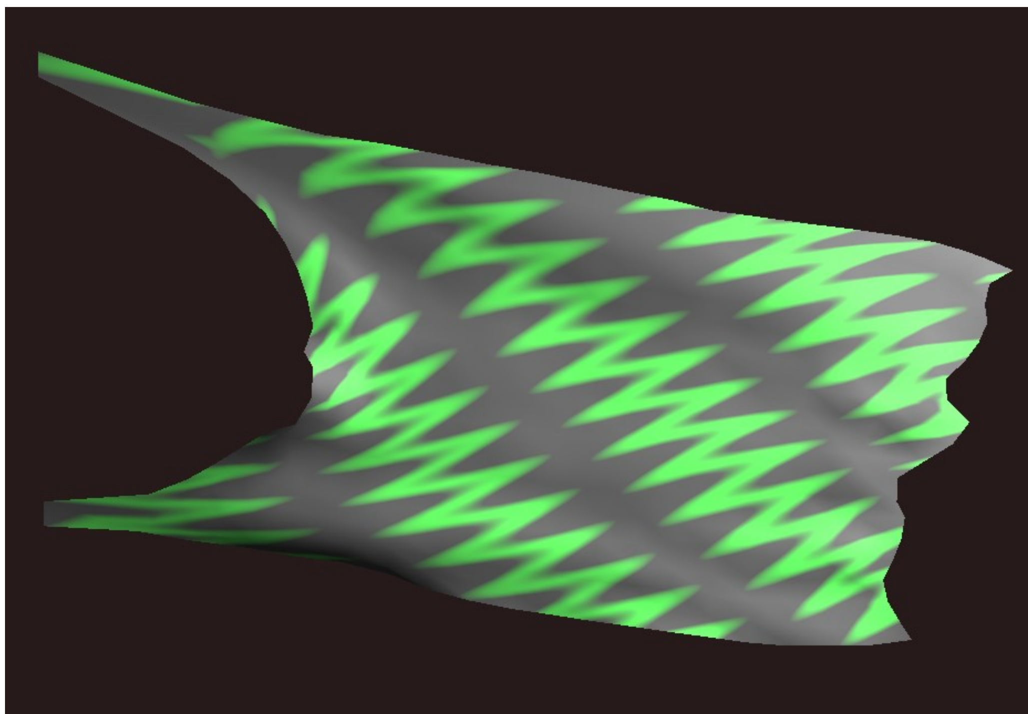


Slika 3.1.4 – Tkanina na kugli, vjetrovito

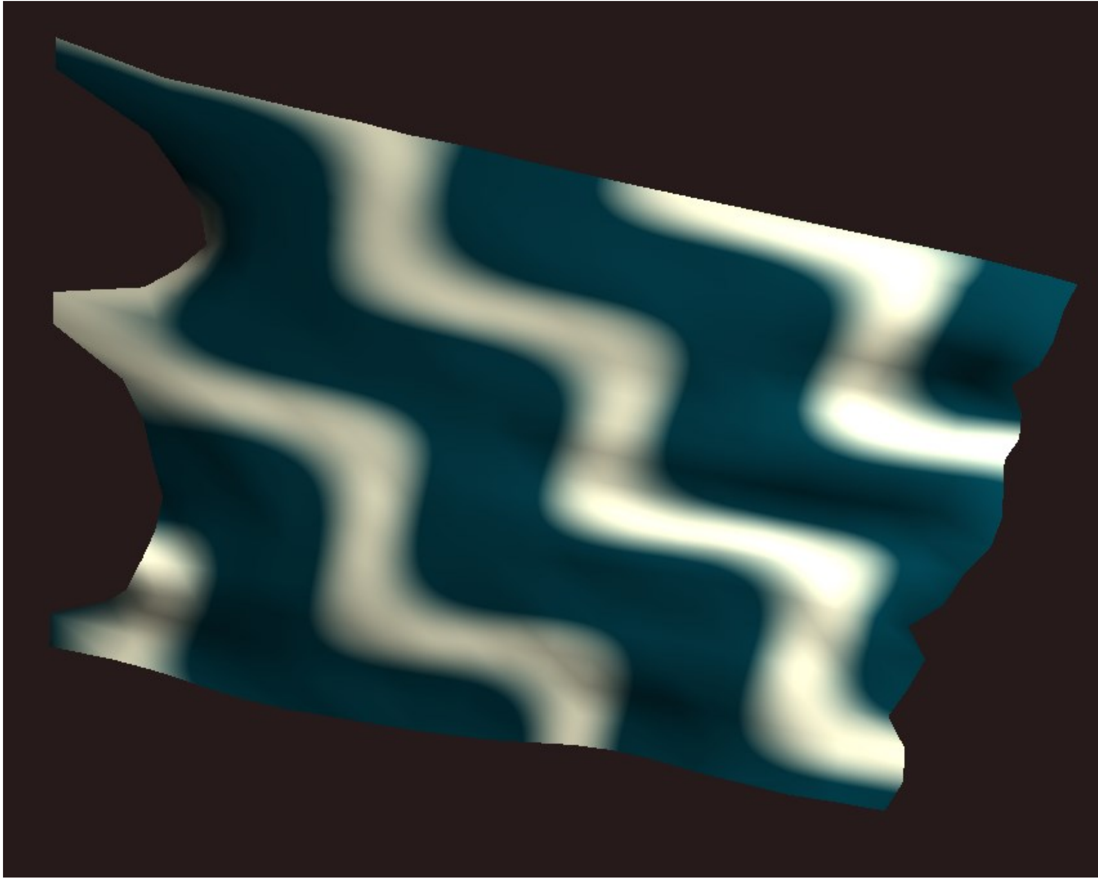
Dodavanjem jakog vjetra tkanina počinje lelujati na vjetru. To je vjerojatno zato što otpor zraka ovisi o normali tkanine. Kada vjetar puše okomito na površinu tkanine, djeluje velikom silom i zabaci tkaninu na drugu stranu. Onda se na drugoj strani opet tkanina okrene prema vjetru i vjetar ponovno djeluje velikom silom, zabacujući ju na originalnu stranu. Taj ciklus se nastavlja i tkanina leluja na vjetru. Tkanina se neko vrijeme drži za kuglu zbog trenja, pa odleti s vjetrom.



Slika 3.1.5 – Zastava, vjetrovito

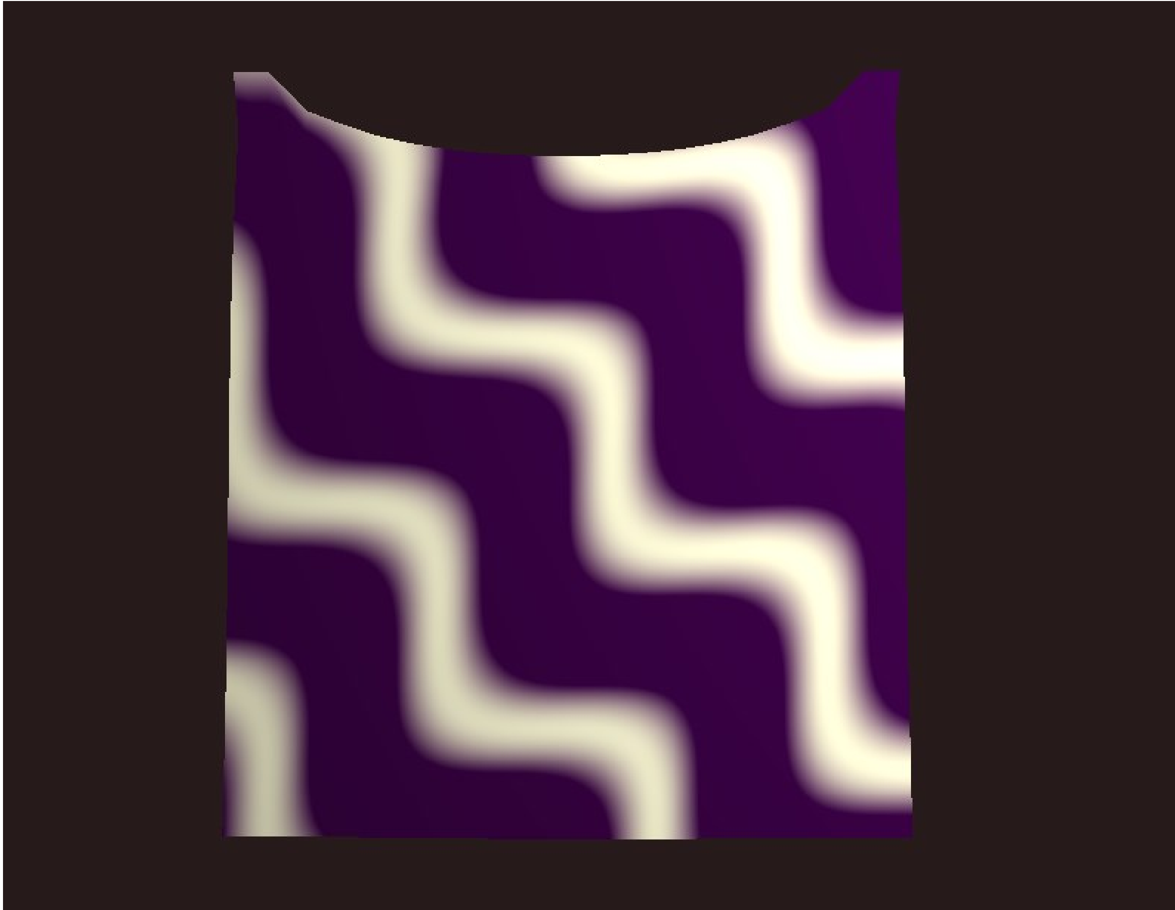


Slika 3.1.6 Zastava pričvršćena dvjema točkama, vjetrovito



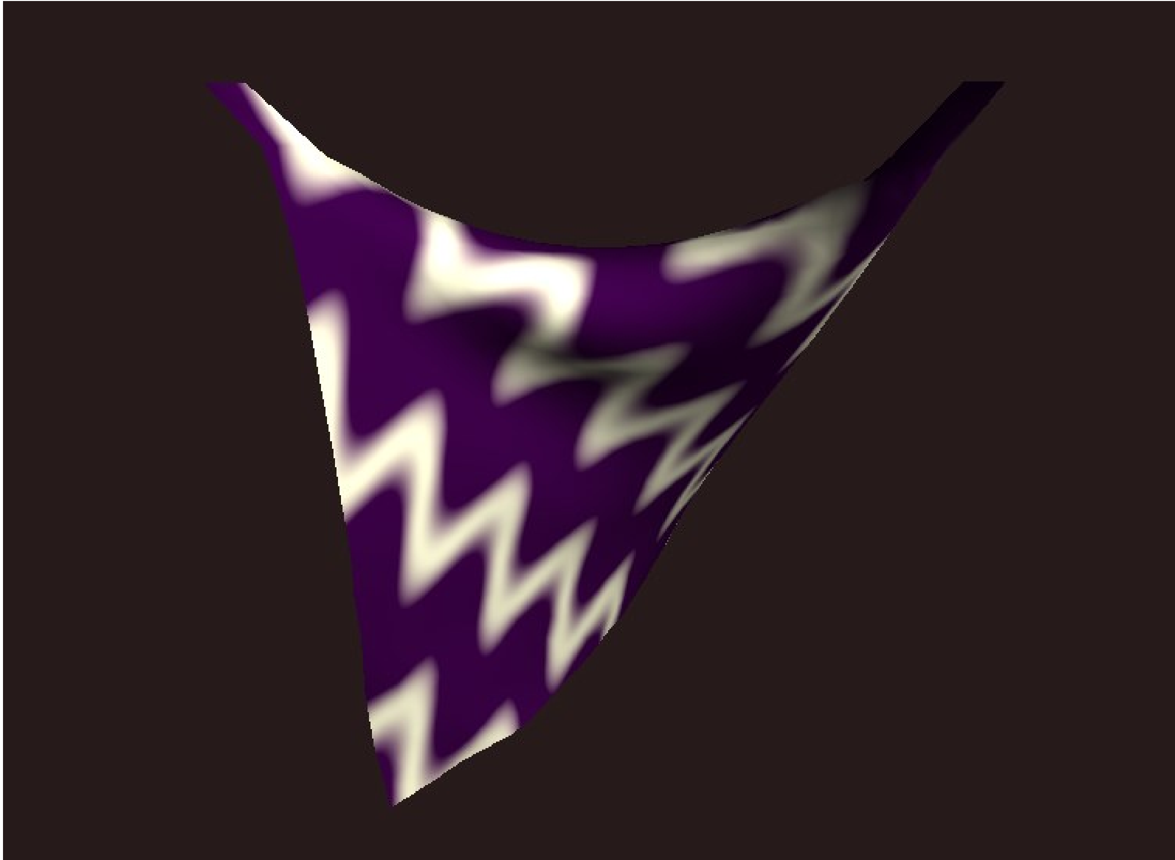
Slika 3.1.7 Zastava pričvršćena trima točkama, vjetrovito

Zastave također lelujaju na vjetru i mali valovi se vide na njihovoj površini. Primjećujemo da se ne događaju neki veći pokreti i promjene, već samo lelujanje male amplitude. To je vjerojatno zato što nedostaje bilo kakav oblik turbulentnog toka zraka. Tok zraka je svugdje isti. Jednom kada se zastava orijentira u tom smjeru, nema puno razloga raditi veće promjene.



Slika 3.1.8 Zastava pričvršćena dvjema točkama pri vrhu, bez vjetra

U ovom slučaju bez vjetra, tkanina je krenula poskakivati, pritom gubeći jako malo energije. Tkanina nije izlazila iz originalne ravnine u kojoj je bila na početku i uvijek bi se vraćala skoro do originalne pozicije. Morao sam drastično povećati koeficijent prigušenja opruge da bi se tkanina ponašala u skladu s očekivanjima i brže došla u stabilnu poziciju kao na slici.



Slika 3.1.9 Zastava pričvršćena dvjema točkama pri vrhu, vjetrovito

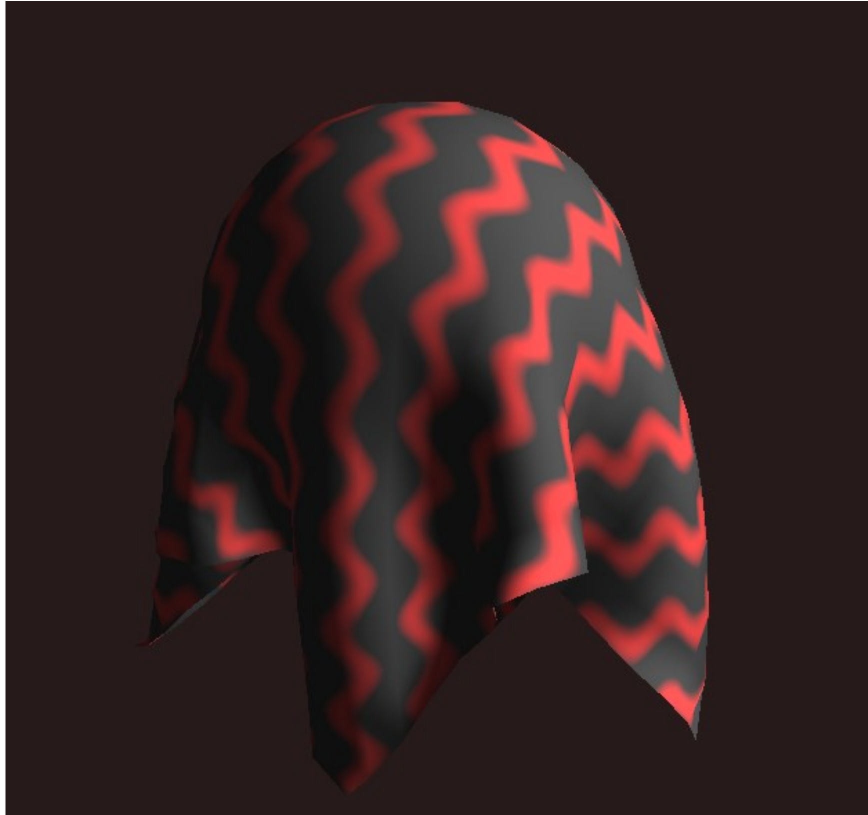
U ovom slučaju vjetar puše u smjeru gledanja kamere. Tkanina se nakon nekog vremena pomaknula u poziciju na slici i ostala do daljnjega u toj poziciji. Jedan kut leluja iza a drugi ostaje naprijed. Ovo ima smisla s obzirom da tkanina u ovoj poziciji više nije toliko okomita na vjetar i stoga vjetar ne djeluje velikom silom na tkaninu. Drugim riječima, ova pozicija je stabilna. No znamo da se stvarne zastave ne ponašaju baš tako u ovakvim situacijama. Stvarne zastave lelujaju većim pokretima i mijenjaju se s jedne strane na drugu. Moguće da bi nekakav oblik turbulentnog toka zraka popravio ovu situaciju i dao tkanini dovoljno energije da izađe iz te stabilne pozicije.

3.2. Konstanta opruge

Promotrimo kako različite vrijednosti konstante opruge utječu na našu simulaciju.

Slika 3.2.1 – Konstanta opruge je 2

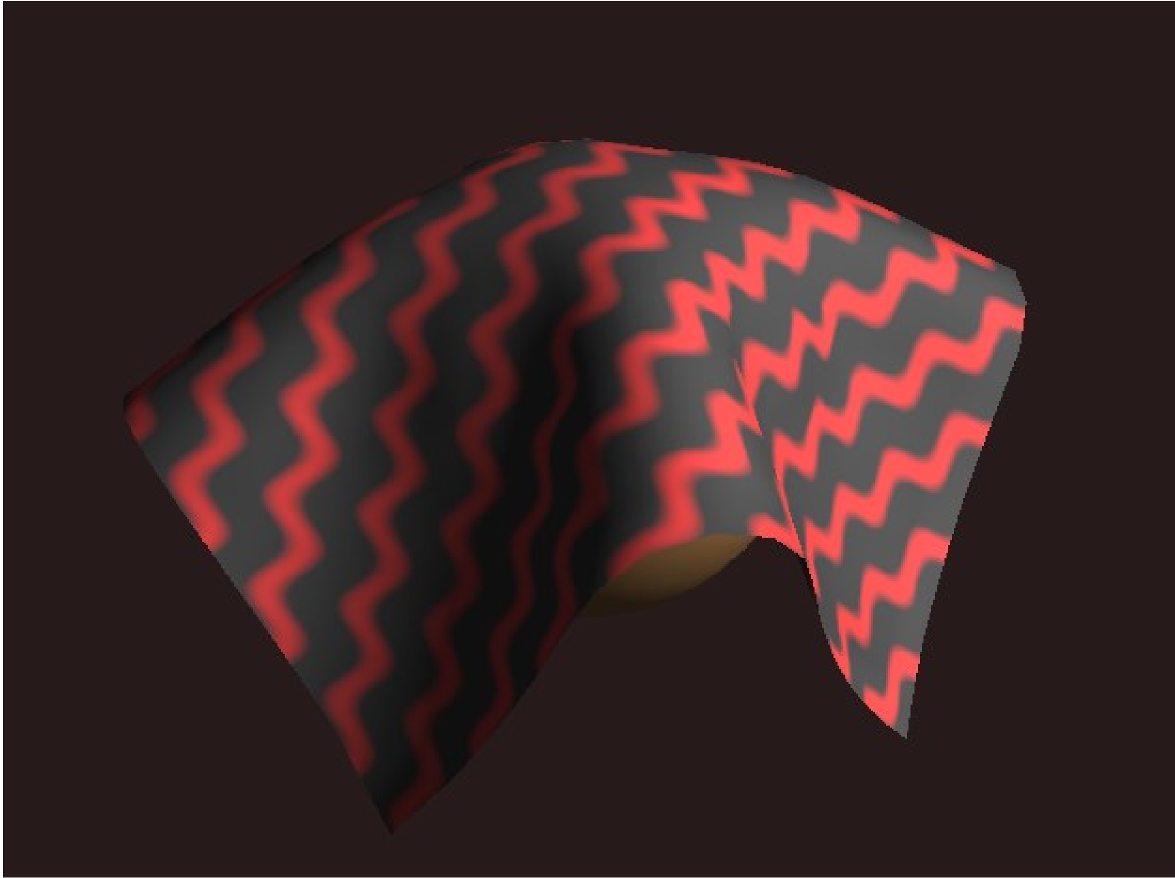
S tako niskom konstantom opruge tkanina nalikuje na nekakvu ljigavu tvar. Izrazito se rasteže zbog gravitacije. Još manja vrijednost konstante opruge uzrokovala bi da se mreža tkanine toliko rastegne da cijela kugla prođe kroz rupu između točaka u mreži.



Slika 3.2.2 – Konstanta opruge je 10

Slika 3.2.3 – Konstanta opruge je 40

Ovo je generalno dobra vrijednost konstante opruge. Daje rezultate u skladu s očekivanjima u puno različitih scenarija.



Slika 3.2.4 – Konstanta opruge je 1000

Ovako visoka konstanta opruge uzrokuje da tkanina više nalikuje na neku vrstu papira (ne želi se savijati u više smjerova istovremeno).

3.3. Dimenzije mreže

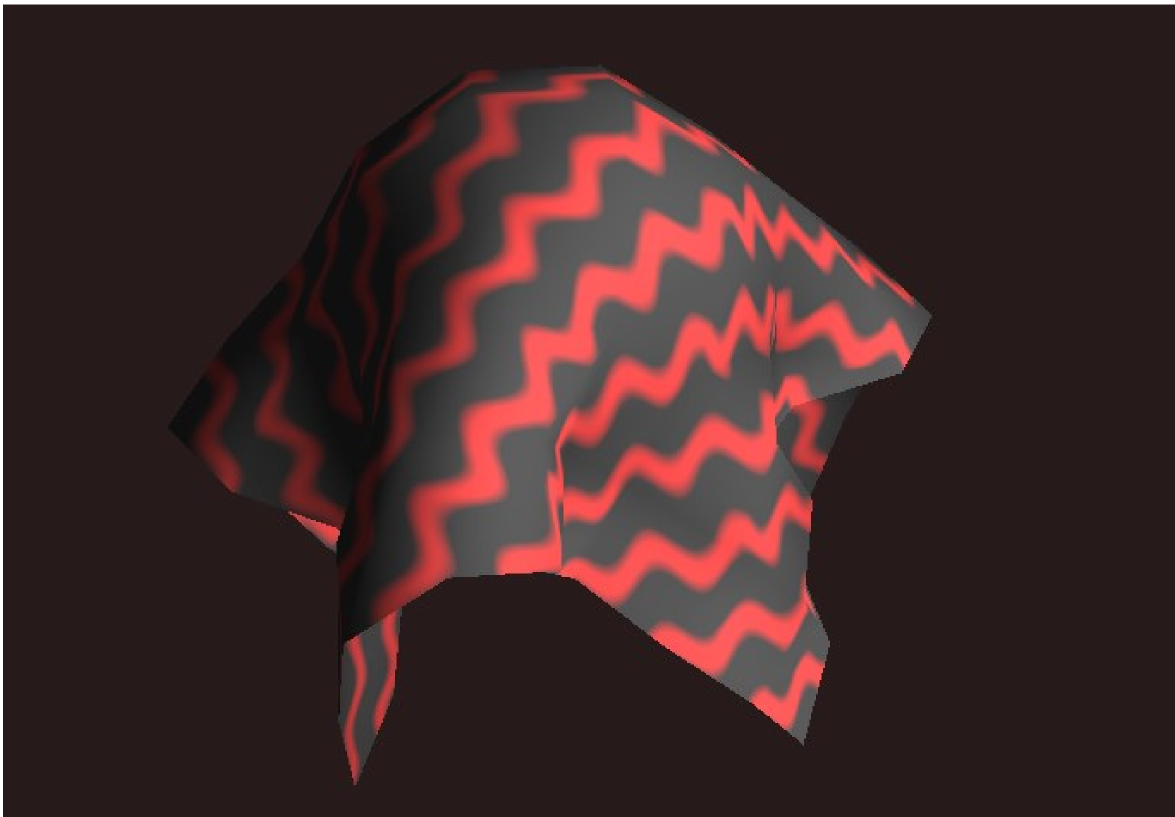
Promotrimo kako različite dimenzije mreže tkanine (broj točaka) utječu na brzinu i izgled simulacije.

Komponente računala korištenog za testove:

CPU: AMD Ryzen 5 5600H

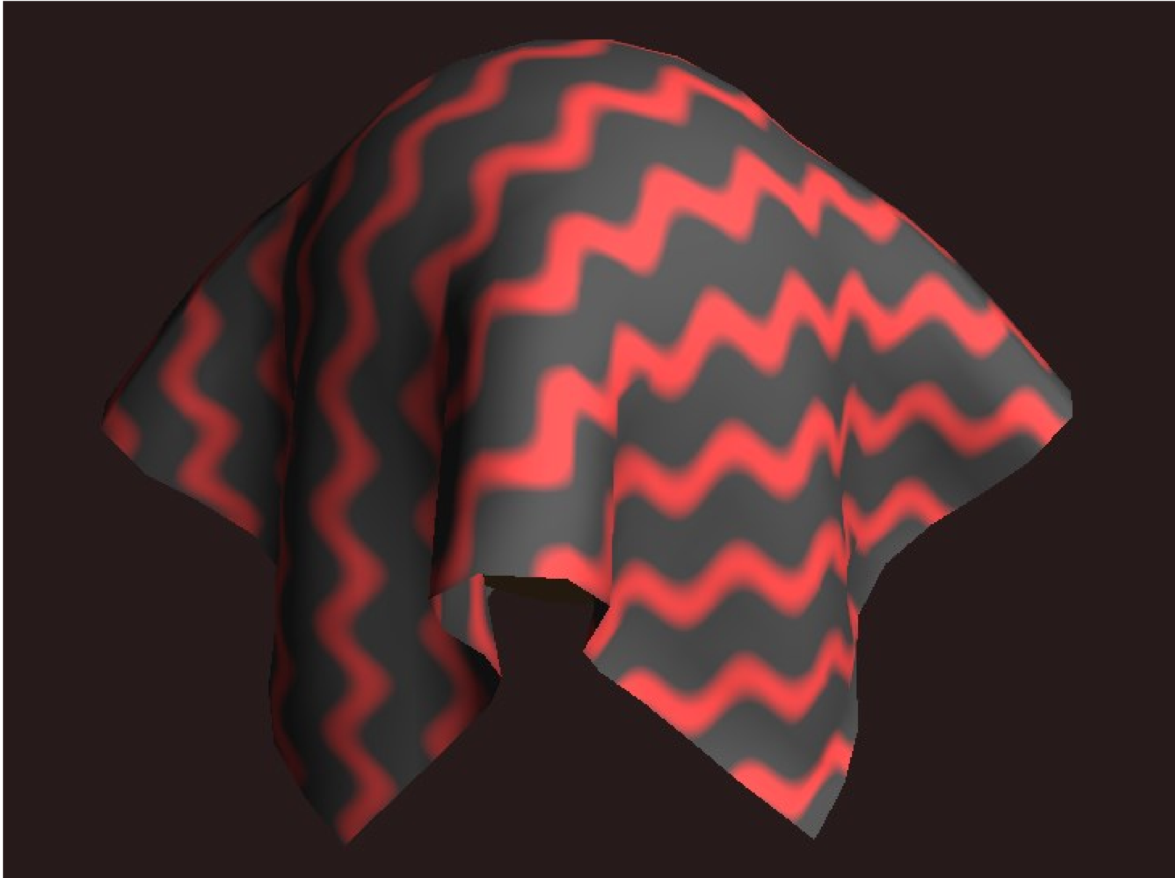
RAM: 16 GB

GPU: NVIDIA GeForce RTX 3050 Ti Laptop GPU



Slika 3.3.1 – Broj točaka 10x10

100 točaka, FPS je 800. Izgled je grub, prikladno za manje tkanine ili tkanine gdje detalji nisu pretjerano bitni. Broj ažuriranih točaka po sekundi je 100x800=80000.



Slika 3.3.2 – Broj točaka 20x20

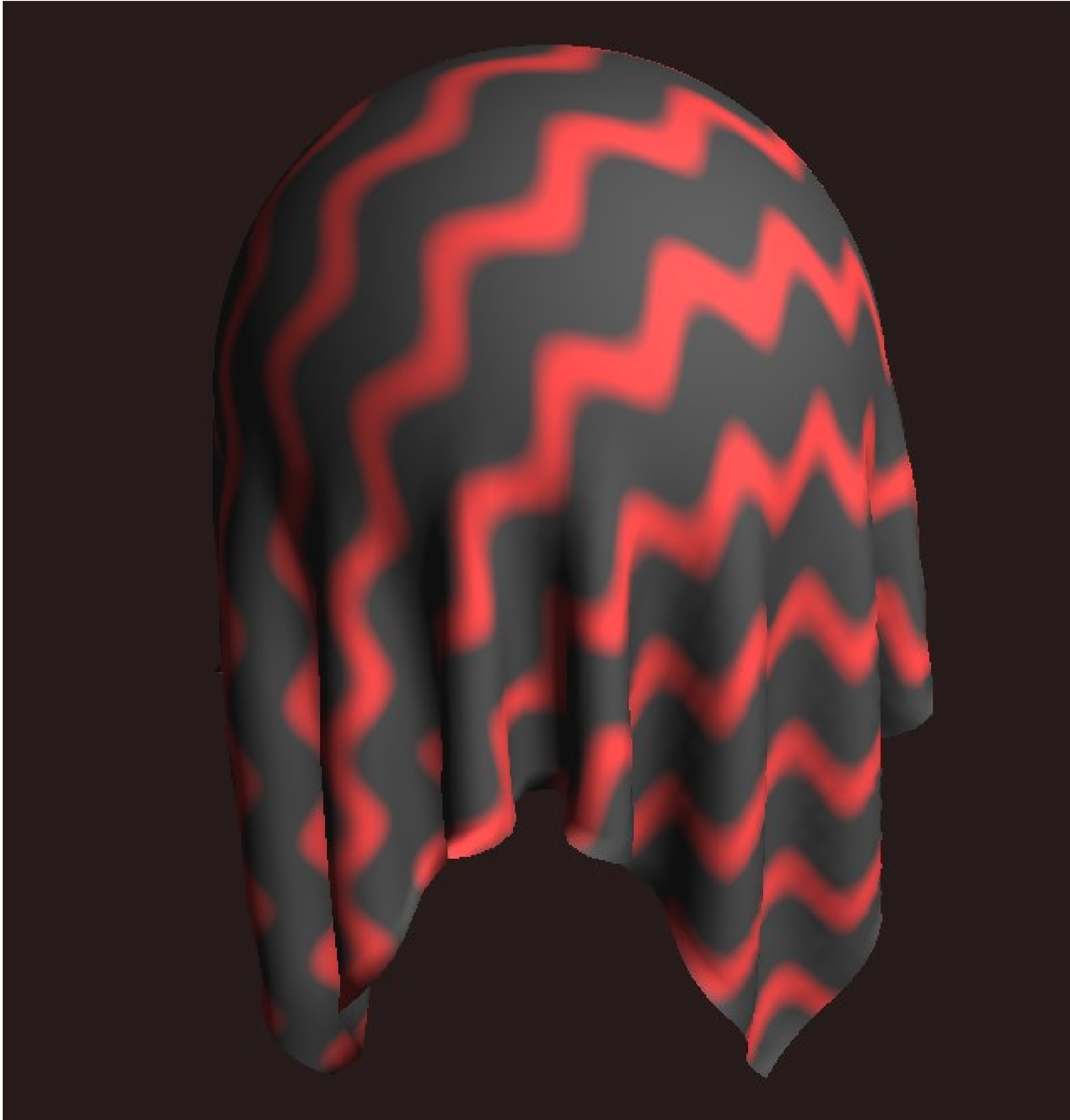
400 točaka, FPS je 190. Dovoljno brzo za primjene u stvarnom vremenu. Dovoljno detaljno da prikaže većinu fenomena tkanine za koje smo zainteresirani. Ove dimenzije mreže su korištene u svim gore prikazanim scenarijima. Broj ažuriranih točaka po sekundi je 400x190=76000.

Slika 3.3.3 – Broj točaka 30x30

900 točaka, FPS je 80. Dovoljno brzo za primjene u stvarnom vremenu. Pomalo usporeno, prikladno za teže ili veće tkanine. Broj ažuriranih točaka po sekundi je 900x80=72000.

Slika 3.3.4 – Broj točaka 40x40

1600 točaka, FPS je 45. Prilično sporo, nije prikladno za primjene u stvarnom vremenu. Količina detalja se povećava. Broj ažuriranih točaka po sekundi je 1600x45=72000.



Slika 3.3.5 – Broj točaka 60x60

3600 točaka, FPS je 20. Vrlo sporo. Velik broj nabora i detalja, izgleda kao finija tkanina. Sveukupna masa tkanine se drastično povećava zbog većeg broja točaka pa je potrebno povećati konstantu opruge da izbjegnemo prekomjerno rastezanje zbog gravitacije. Broj ažuriranih točaka po sekundi je 3600x20=72000.

FPS je obrnuto proporcionalan s brojem simuliranih točaka. Drugim riječima, broj ažuriranih točaka po sekundi je konstantan (oko 75000), što je za očekivati.

3.4. Moguća poboljšanja

Najveća prilika za nadogradnju bila bi paraleliziranje ovog algoritma uz pomoć računskog sjenčara (compute shader). Algoritam bi se mogao dobro paralelizirati s obzirom da su izračuni većinom nezavisni za svaku česticu. To bi omogućilo korištenje mreža velikih dimenzija u stvarnom vremenu unatoč kompleksnosti simulacije.

Još jedno moguće poboljšanje bilo bi simuliranje ili emuliranje turbulentnog toka zraka. To bi omogućilo realističnije ponašanje tkanine u zraku.

Jedan problem koji se događao tijekom testiranja je da bi tkanina prošla kroz samu sebe. U većini slučajeva se nije događalo ili bi bilo vrlo suptilno s obzirom da je korišteno dosta pregibnih veza koje sprječavaju prekomjerno savijanje, ali u nekim slučajevima gdje se udaljeniji dijelovi tkanine dodiruju, došlo bi do prolaska tkanine kroz samu sebe. Implementacija algoritma za sudar tkanine sa samom sobom bilo bi dobro poboljšanje simulacije.

Zaključak

Iz rezultata testova smo vidjeli da simulacija masa i opruga ove razine kompleksnosti (20 najbližih susjeda, sudarači, trenje, otpor zraka ovisan o normali tkanine) koja se izračunava na jednoj dretvi nije prikladna za primjene u stvarnom vremenu ako se radi o većim mrežama s puno nabora i detalja. Međutim, ako se radi o manjim mrežama koje ne moraju imati puno malih nabora, ovaj model nudi praktičan način za implementirati simulaciju tkanine za primjene u stvarnom vremenu poput računalnih igara.

Dodatna korist ovog modela je ta što se može implementirati korištenjem vrlo intuitivnih i jednostavnih fizičkih jednadžbi od kojih se većina uče u osnovnoj školi. Upravo zbog toga što su osnovni elementi ove simulacije vrlo intuitivni, kombiniranje tih elemenata u sveukupnu simulaciju nam omogućuje da bolje razumijemo cjelokupno ponašanje tkanine u raznim situacijama. Iz jednostavnih pravila koja vrijede za dijelove nastaje kompleksno ponašanje cjeline.

Literatura

[1] *Cloth Simulation using Mass-Spring System,* UC Irvine, Poveznica: https://ics.uci.edu/~shz/courses/cs114/docs/proj3/index.html; pristupljeno 8. rujna 2024.

[2] *Spring mass damper theory*, ReStackor, Poveznica: https://restackor.com/physics/response/spring-mass-damper; pristupljeno 8. rujna 2024.

[3] *Drag Forces*, Lumen Learning, Poveznica: https://courses.lumenlearning.com/suny-physics/chapter/5-2-drag-forces/; pristupljeno 8. rujna 2024.

[4] *Verlet integration,* Algorithm Archive, Poveznica: https://www.algorithm-archive.org/contents/verlet_integration/verlet_integration.html; pristupljeno 14. lipnja 2024.

[5] *How to compute mesh normals*, NYU Media Research Lab, Poveznica: https://mrl.cs.nyu.edu/~perlin/courses/fall2002/meshnormals.html; pristupljeno 8. rujna 2024.

Sažetak

Simulacija i vizualizacija modela tkanine

Obrađena je teorija različitih fizikalnih modela koji se koriste u simulaciji tkanine računalom. Navedene su sile koje se mogu uračunati tijekom simulacije tkanine, kao i njihove formule. Objašnjene su neke metode numeričke integracije pozicije koje se mogu koristiti pri implementaciji modela tkanine. Istražena je implementacija modela masa i opruga koristeći Verlet integraciju uz dodatak sudarača, trenja i otpora zraka. Objašnjene su uloge najbitnijih razreda i funkcionalnosti najbitnijih metoda. Opisani su sjenčari korišteni u vizualizaciji tkanine. Postavljeni su razni scenariji od interesa i razmotreno je ponašanje simulacije u tim scenarijima. Posebna pozornost je obraćena na utjecaj konstante opruge i broja točaka u mreži tkanine. Razmotrena su potencijalna poboljšanja simulacije.

Ključne riječi: simulacija, vizualizacija, simulacija tkanine, računalna grafika, računalno modeliranje, C++, OpenGL

… # Summary

Simulation and Visualization of Cloth Model

The theory of different physical models used in cloth simulation is discussed. The forces that can be accounted for during cloth simulation, along with their formulas, are listed. Some methods of numerical integration of position, which can be used in the implementation of cloth models, are explained. An implementation of the mass-spring model using Verlet integration, along with the addition of colliders, friction, and air resistance, is explored. The roles of the most important classes and the functionalities of the most important methods are explained. The shaders used for visualizing cloth are discussed. Various scenarios of interest are set up, and the behavior of the simulation in those scenarios is examined. Special attention is given to the impact of the spring constant and the number of points in the cloth grid. Potential improvements to the simulation are considered.

Keywords: simulation, visualization, cloth simulation, computer graphics, computer modelling, C++, OpenGL

Privitak

Implementaciju je moguće pronaći na poveznici:

https://github.com/MarinLovrinovic/ClothSim