Andrea Passini,
RTGP Project

Cloth Simulation

Introduction

The goal of this project is to implement a cloth using different techniques to emulate the connection between each fiber of a real cloth.
The cloth will be made by a grid of particles connected with different types of constraint, to test the visual effect and their efficiency.
In order to achieve that it has been implemented a small dynamic system to apply forces to the particles, using the Verlet's numerical integration method, and a collision system, that will detect the collision of the cloth with some spheres and planes, and will calculate and apply the response of those collision using the PBD (particle base dynamics) method that works well with Verlet's.

Particles

Each particle will be a dynamic Newtonian object. Dynamics is the field of study where the motion is seen as the result of forces.
Each particle will be affected by forces, those forces will cause a change in position of the particle.
To do that we can divide the problem in 2:

1. Force cumulate in a particle will cause a change in accelleration, following the **Newton's second law**:

   $$f = m \cdot a$$

   That inverted will gives us the formula to calculate the acceleration cause by that force

   $$a = f / m$$

2. Acceleration will cause a change in position, (acceleration is the second derivative of the position). In order to do that an integration method is needed, the one chosen is the **Verlet's integration method.**

Verlet's integration method

With this integration method, unlike in the Forward Euler, Symplectic Euler or in the Leapfrog integration methods, the state of the particles does not contains its velocity, instead is state is made by:

Particle State

- **Current position "$Pnow$"**
- **Previous position "$Pold$"**

The velocity is kept implicit but can be calculated:

$$v = (Pnow - Pold) / dt$$

This integration method has been chosen because of:

- It accuracy, being a second order method:
  - Per-step error is linear with the dt
  - Accumulated error is in the order of dt^2 (accumulated error tends to be more important in a dynamic simulation)

- Since we save the previous and the actual position, a change in the actual position will result in a change in velocity. So we will be able to directly affect the position of a particle, without needing to add a force, without breaking the dynamic simulation.

Verlet's Step

For each particle, we need to calculate the next position based on the accumulated force applied to the particle.

$$\vec{a} \leftarrow \vec{f} / m$$
$$Pnext \leftarrow 2Pnow - Pold + \vec{a} \cdot dt^2$$

$$Pold \leftarrow Pnow$$
$$Pnext \leftarrow Pnow$$

Velocity Damping

In reality the total energy of a system cannot increase, a bit of energy will be dissipated, when attrition dissipates energy into heat, or sound waves are produced, decreasing the total energy of the system.
In a simulation, integration errors may cause the total energy of the system to increase over time, compromising the stability of the simulation.
In order to avoid this problem we have 2 options:

- **Attrition**
  We can introduce attrition as forces in our system, opposed to the current velocity direction
- **Velocity damping**
  A big simplification is to simplify attrition by reducing the velocity vector by a fixed proportion

For the sake of simplicity, the second option has been chosen, and applied inside the Verlet's Step:

Given a ***C****damp* factor (value around 0.98) obtained as

$$Cdamp \leftarrow 1 - dt \cdot Cdrag$$

The Verlet's step will be:

$$\vec{a} \leftarrow \vec{f} / m$$
$$Pnext \leftarrow (1 + Cdamp)Pnow - Cdamp \cdot Pold + \vec{a} \cdot dt^2$$
$$Pold \leftarrow Pnow$$
$$Pnext \leftarrow Pnow$$

```
#define FIXED_TIME_STEP (1.0f / 60.0f)
#define FIXED_TIME_STEP2 (FIXED_TIME_STEP * FIXED_TIME_STEP)
#define DAMPING 0.02f
void PhysicStep()
{
    glm::vec3 now_pos = pos;
    glm::vec3 accel = this->force/mass;
    pos = ((2.0f - DAMPING) * now_pos) - ( (1.0f - DAMPING) * old_pos) + (accel * FIXED_TIME_STEP2);
    old_pos = now_pos;

    this->force = glm::vec3(0.0f);
}
```

Fixed Time Step

One of the main requirements of Verlet's Integration Method is to keep the delta time fixed. One problem emerges: the rendering loop does not have a fixed delta time, once the gpu fills up the buffer with the new image, the buffer switch happens and the loop starts again. In order to keep the delta time fixed for the physic update and the delta time free for the rendering update: physic time and rendering time will be kept separated, the physic time will be incremented with a fixed amount, the rendering time will be the time of the application. The increment of the physic time will happen based on the comparison with the rendering time.
When the physic clock is behind the rendering clock, when its value is less than the one of the rendering clock, it will be incremented by a fixed amount of time (1/60 or 1/30 depending on the physic framerate chosen by the user). When the physic time is incremented, the physic simulation will execute its step, calculating the results of the forces applied on each particle and updating their positions.

```
while(!physicsSimulation.isPaused &&  currentTime > physicsSimulation.getVirtualTIme()){
    physicsSimulation.FixedTimeStep();
    cloth.PhysicsSteps(activeScene);
    physIter++;

    if(physIter > maxIter){
        std::cout << "Physics Simulation lagging " << std::endl;
        physicsSimulation.SynchVirtualTime(currentTime);
        break;
    }
}
```

If even after incrementing the physic time, this is still behind the rendering time, it is possible to increment it again, and execute the physic step. This procedure can be repeated for a limited number of times. (maxIter)

Constraints

The particles in the cloth will be attached one to another by a constraint.
The constraints have been implemented in 2 different ways:

- **Positional constraint**
  Exploiting the Verlet's Numerical Integration Method state, we can reposition each particle without adding some extra forces.
  This approach is usually called Position Based Dynamics and can be used also to simulate rigid bodies.

  Given the particles: $a,\ b$
  Given their positions: $Pa,\ Pb$
  Given a rest length (the length that we want to keep between particles of the constraint): d

$$
\begin{aligned}
v &\leftarrow Pa - Pb \\
currDist &\leftarrow v.length \\
v &\leftarrow v \,/\, currDist \\
delta &\leftarrow currDist - d \\
Pa &\leftarrow Pa + (0.5 \cdot delta) \cdot v \\
Pb &\leftarrow Pb - (0.5 \cdot delta) \cdot v
\end{aligned}
$$

  The factor 0.5 is established assuming both particles have equal masses, so we will move each of them halfway.

```
void satisfyPositionalConstraint(float K)
{
    glm::vec3 correctionVector = CalculateCorrectionVector(K);

    this->p1->offsetPos(correctionVector);
    this->p2->offsetPos(-correctionVector);
}
```

```
glm::vec3 CalculateCorrectionVector(float K){
    glm::vec3 p1_to_p2 = this->p2->getPos() - this->p1->getPos(); // vector from p1 to p2
    float current_distance = glm::length(p1_to_p2); // current distance between p1 and p2

    if(cuttable){···

    p1_to_p2 /= current_distance;

    float deltaDistance = current_distance - rest_distance;

    glm::vec3 correctionVector = K * deltaDistance * p1_to_p2;
    return correctionVector;
}
```

If we want a more general solution we could calculate the factor like this:
Given the masses of the particles: $Mb,\ Ma$

$$
\begin{aligned}
v &\leftarrow Pa - Pb \\
currDist &\leftarrow v.length \\
v &\leftarrow v\,/\,currDist \\
delta &\leftarrow currDist - d \\
Pa &\leftarrow Pa + (Mb/(Ma+Mb)\cdot delta)\cdot v \\
Pb &\leftarrow Pb - (Ma/(Ma+Mb)\cdot delta)\cdot v
\end{aligned}
$$

This reposition will be applied to each particle after the Verlet's Step.

- **Spring or Physical constraint**
  A spring is simulated applying the Hooke's law:

  Given rest length of the spring: l
  Given stiffness of the spring: k

$$
\begin{aligned}
f_a &\leftarrow k\,(||Pb - Pa|| - l)\frac{Pb-Pa}{||Pb-Pa||} \\
f_b &\leftarrow -f_a
\end{aligned}
$$

  This force will be added to the cumulated force of the particle and applied, during the Verlte's Step.

```cpp
void satisfyPhysicsConstraint(float K)
{
    glm::vec3 correctionVector = CalculateCorrectionVector(K);

    this->p1->addForce(correctionVector);
    this->p2->addForce(-correctionVector);
}
```

```cpp
glm::vec3 CalculateCorrectionVector(float K){
    glm::vec3 p1_to_p2 = this->p2->getPos() - this->p1->getPos(); // vector from p1 to p2
    float current_distance = glm::length(p1_to_p2); // current distance between p1 and p2

    if(cuttable){···

    p1_to_p2 /= current_distance;

    float deltaDistance = current_distance - rest_distance;

    glm::vec3 correctionVector = K * deltaDistance * p1_to_p2;
    return correctionVector;
}
```

- **Spring and Friction or Advanced Physical constraint**
  In addition to the force of spring, calculated using the method before, friction is included as part of the cumulated force of the particle.
  A dissipative force, given a Damping factor k_d, that want to slow down the elongation or the shrinking of the spring

  Given a damping factor k_d

$$\hat{d} = \frac{p_b - p_a}{\|p_b - p_a\|}$$

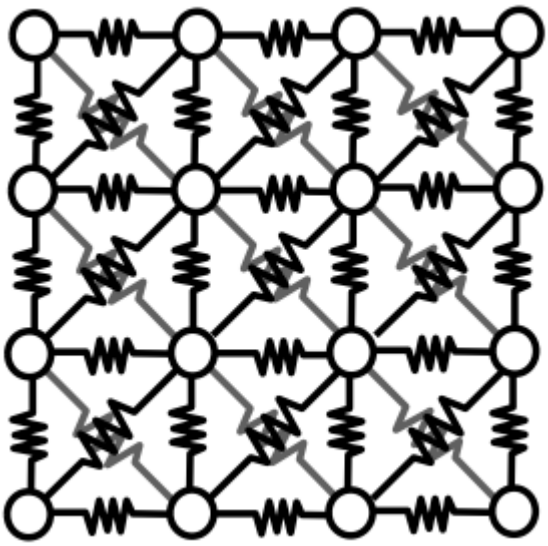
$$\vec{f_a} = k_d(\hat{d} \cdot (\vec{v_b} - \vec{v_a}))\,\hat{d}$$

```
void satisfyAdvancedPhysicalConstraint(float K, float U, float deltaTime){
    glm::vec3 correctionVector = CalculateCorrectionVector(K);

    this->p1->addForce(correctionVector);
    this->p2->addForce(-correctionVector);

    glm::vec3 springFrictionVector = CalculateSpringFrictionVector( U, deltaTime);

    this->p1->addForce(springFrictionVector);
    this->p2->addForce(-springFrictionVector);
}
```

```
glm::vec3 CalculateSpringFrictionVector(float dampingFactor, float deltaTime){
    // dampingFactor(d*(v2-v1))*d
    glm::vec3 p1_to_p2 = this->p2->getPos() - this->p1->getPos(); // vector from p1 to p2
    float current_distance = glm::length(p1_to_p2); // current distance between p1 and p2

    p1_to_p2 /= current_distance;   // Normalize

    glm::vec3 v_1 = (this->p1->pos - this->p1->old_pos) / deltaTime; // Speed as the deltaDistance pos in time
    glm::vec3 v_2 = (this->p2->pos - this->p2->old_pos) / deltaTime; // Speed as the deltaDistance pos in time
    glm::vec3 v2_minus_v1 = v_2 - v_1;

    glm::vec3 correctionVector = dampingFactor * (p1_to_p2 * (v2_minus_v1)) * p1_to_p2;
    return correctionVector;
}
```

Cloth

The cloth is made by a grid of particles connected by constraints.



```cpp
int dim; // number of particles in "width" direction
// total number of particles is dim*dim

std::vector<Particle> particles; // all particles that are part of this cloth
std::vector<Constraint> constraints; // alle constraints between particles as part of this cloth
```

Level of Constraint

A particle can be attached to every particle at distance 1, 2, 3… ,depending on how we set-up the parameter from the UI, with every level of connection the cloth becomes more stiff and resistant to the elongation or shrinking.

In addition to the level of constraint we can also set how many times we want the cloth to recalculate the effect of the constraint on the particle, allowing us to fine tune the behavior of the cloth considering also the framerate.

```cpp
for(int x=0; x < dim; x++)
{
    for(int y=0; y < dim; y++)
    {
        /*
        // Constraints from top left
        // * ---
        // | \
        // |  \
        */
        for(int i = 1; i <= this->constraintLevel; i++){
            if(y+i < dim) makeConstraint(getParticle(x, y, dim), getParticle(x, y+i, dim), particleDistance * i, cuttingDistanceMultiplier);
            if(x+i < dim) makeConstraint(getParticle(x, y, dim), getParticle(x+i, y, dim), particleDistance * i, cuttingDistanceMultiplier);
            if(y+i < dim && x+i < dim) makeConstraint(getParticle(x, y, dim),
                                                      getParticle(x+i, y+i, dim),
                                                      particleDistance*glm::sqrt(2.0f) * i,
                                                      cuttingDistanceMultiplier);
        }
    }
}
```

Rendering

In order to render the cloth we need to pass the array of triangles and their attributes to the gpu.
To build the triangles, for every square of particle triangles are created.
The main attributes (per particle) that the shader will need in order to render the cloth are:

- Position
- Normal
- Base color
- Force

For every triangle that we created a normal is calculated by cumulating the normal of the triangles that this particle is in. The normal of the triangle is calculated using the cross product of the 2 edges of the triangle (topl-eft and top-right with top-left and bottom left).

```
void MakeTriangleFromGrid(){
    indices.clear();
    for(int x = 0; x < dim-1; x++)
    {
        for(int y=0; y < dim-1; y++)
        {
            if(getParticle(x, y, dim)->renderable &&
                getParticle(x, y+1, dim)->renderable &&
                getParticle(x+1, y, dim)->renderable)
            {
                indices.push_back((x)*dim +y);
                indices.push_back((x)*dim +(y+1));
                indices.push_back((x+1)*dim +(y));
            }

            if(getParticle(x+1, y, dim)->renderable &&
                getParticle(x, y+1, dim)->renderable &&
                getParticle(x+1, y+1, dim)->renderable)
            {
                indices.push_back((x+1)*dim +(y));
                indices.push_back((x)*dim +(y+1));
                indices.push_back((x+1)*dim +(y+1));
            }
        }
    }
}
```

```
void UpdateNormals(){
    std::vector<Particle>::iterator particle;
    for(particle = particles.begin(); particle != particles.end(); particle++)
    {
        particle->resetNormal();
    }

    for(int x = 0; x < dim-1; x++)
    {
        for(int y=0; y < dim-1; y++)
        {
            glm::vec3 normal;

            normal = CalculateNormalTriangle(getParticle(x, y,   dim), getParticle(x+1,   y,   dim), getParticle(x,   y+1, dim));
            getParticle(x,   y,   dim)->addToNormal(normal);
            getParticle(x+1, y,   dim)->addToNormal(normal);
            getParticle(x,   y+1, dim)->addToNormal(normal);
        }
    }
}
```

Shader

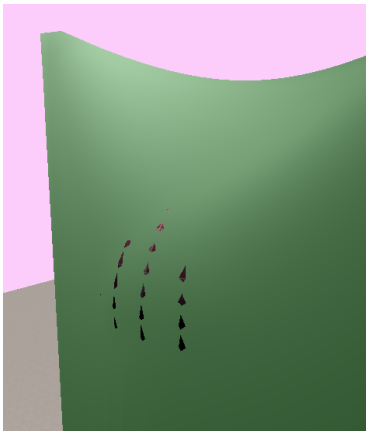
The shader used for this project are:

- **Color Shader**:
  Using the GGX illumination formula with the color set at the creation of the cloth.
- **Phong Force Shader**
  Using the Blinn-Phong illumination formula with the value of the force affecting that specific particle.
- **GGX Force Shader**
  Using the GGX illumination formula with the value of the force affecting that specific particle.

To increment the visual difference of the 2 force shader, in the Blinn-Phong the force is normalized before being used in the illumination formula, causing it to have lighter colors.

Collisions

The cloth can detect and react to collisions and are solved using the PBD (Particle based Dynamics approach used also in the positional constraints).
The collisions in this project can be tuned with 2 parameters, the number of times the cloth will recalculate the detection-response of the collision and at which distance we want the cloth to be in respect to the object we have collided with.
This last parameter is useful to avoid an unpleasant visual effect that happens when the particle is not inside the sphere but, since there is a bit of space between each particle, it can look like a section of the cloth is inside the sphere.



For the sake of this project the collision will be considered only as a 1-way collision, a completely elastic collision with an object that has an infinite mass. In simpler words, during the collision the particles of the cloth will be repositioned and the other object colliding with it will not be affected by the collision.

Collision with a sphere

The collision with a sphere is easy to detect and solve:

- **Detection**
  A particle is inside the sphere if the distance between the sphere and the particle is smaller than the radius of the sphere
- **Response**
  Reposition the particle based on the difference of the actual distance and the radius. (The radius is multiplied by the parameter to avoid the visual effect described before).

```
void SphereCollision(const glm::vec3 centerWorld, const float radius){
    glm::vec3 v = pos - centerWorld;

    float l = glm::length(v);

    if (l < (radius * COLLISION_OFFSET_MULTIPLIER)) // if the particle is inside the ball
    {
        // project the particle to the surface of the ball
        this->offsetPos(glm::normalize(v) * (((radius * COLLISION_OFFSET_MULTIPLIER)-l)));
    }
}
```

Collision with a plane

A plane is stored as a position of a point on the plane and the normal of the plane.

- **Detection**
  To detect if a particle is under the plane we can exploit the dot-product:
  - Calculate the distance between the particle and the point on the plane
  - Dot product of the distance with the normal of the plane,
    - < 0, the particle is under the surface of the plane, collision
    - > 0, the particle is over the surface

- **Response**
  We offset the particle by a vector with the same direction of the normal with a magnitude of the inverse of the result of the dot product, used for detection.

```
void PlaneCollision(const glm::vec3 normal, const glm::vec3 pointOnPlane){
    glm::vec3 distance = this->pos - pointOnPlane;
    float dot = glm::dot(distance, normal);

    if(dot <= 0.0f) // under the plane
    {
        glm::vec3 reposition = normal * (-1 * dot * COLLISION_OFFSET_MULTIPLIER);
        this->offsetPos(reposition);
    }
}
```

Scenes

There are 3 scenes, each with a different goal:

- **Scene 1**: you will be able to move a sphere and make it collide with a cloth.
- **Scene 2:** you will be able to move the cloth and some spheres near it, to test it like a coat.
- **Scene 3:** you will be able to move a sphere and make it collide with a cloth but this time the sphere will be able to cut a hole in the cloth based on the cut length parameter, controlled by the user.
  The constraint evaluator will also check if the distance between the particles is larger than the cutting distance. Once a hole needs to be cut, the constraint connected to that particle will be eliminated, the particle will no longer be rendered and new triangles need to be created and passed to the GPU.

Controls

Since the objective of the project is to implement and evaluate the different techniques used to simulate a cloth, there are a lot of parameters that can be tweaked in order to test the cloth. Some of these parameters need to be clamped between some lower and upper bound otherwise they will make the simulation explode, unexpected behavior will happen and most likely the program will crash.
I will report the list of parameters and the range in which the value needs to stay.

These value works for me using a windows desktop computer with these components:

- GPU Nvidia GTX 1070
- CPU AMD Ryzen 5 1600x
- 16 GB RAM

Parameters:

- **Cloth dim:** the size of the particle matrix. Over 50 will really slow the simulation down.
- **Particle offset:** the distance between each particle, the value used in the video is 0.15, if too small the cloth will end up being very small and, if enlarged its dimension the simulation will slow down. Having a value too large may cause some issue in the rendering, the constraints will be very evident, and in the collision detection, a small sphere may pass through the cloth without colliding with any particle.
- **Shader Selector:** allows you to switch between the 3 shaders
- **Scene Selector:** choose between the 3 scenes.
- **Particle mass:** having a value of 0.1 is suggested
- **Gravity force**
- **Type of constraint:** choose between the 3 type of constraint explained before
- **K:** elastic coefficient, its value may vary a on based on the type of constraint:
    - **Positional**: has to be at 0.5, otherwise the formula is not respected since every particle have the same mass
    - **Spring:** a value of 15 is suggested with a constraint iteration of 10 and constraint level of 2
    - **Spring and Friction:** a value of 10 is suggested with constraint iteration of 5 and constraint level of 2, (the damping factor is suggested at 0.1)
- **Constraint iteration:** how many time the constraints or springs need to be evaluated
- **Collision Iteration:** how many times the collision detection and response need to be evaluated.

Conclusion

Here is a link to the video of the tests conducted on the cloth using different constraints and parameters https://youtu.be/Kp4sDx6lbL0

Efficiency
One of the main parameters to consider while choosing the right type of constraint for your application, is the efficiency, in simpler words: does the framerate is high and stable during the simulation?
In this table will be reported the frame per second of each type of cloth in the different scenes.

The measurements are taken with:

- Cloth dimension of 30x30
- Particle distance of 0.15

The other settings are set to the optimal value specified before.

Changing shader will not affect that much the simulation, only around 5 frames will be lost

	Positional Constraint	**Spring**	**Spring and Friction**
Scene 1	**1000**, **900** when a collision happens	**950**, **900** when a collision happens	**700**, **650-600** when a collision happens
Scene 2	**800** when the entire cloth moves	**800-750** when the entire cloth moves	**600-550** when the entire cloth moves
Scene 3	**950**, **850-900** when cut happens	**950**, **850-900** when cut happens	**700**, **600** when cut happens

Visual effect
Another aspect to choose the right cloth type can be the visual appearance of the cloth itself. Changing type of constraint results in different behavior of the cloth and how much it is believable during collisions, enlargements and shrinkings.

- The **Positional** cloth seems like to be the least believable, its movement looks very stiff and the constraints sometimes are visible.
- The **Spring** cloth has a smoother look, the constraints are not visible and the elongation and shrinking looks more natural, but, when a big collison happens, on the surface a lot of small waves appear resulting in a not so believable behavior.
- The **Spring and Friction** cloth has the best and more believable look. Elongation and shrinking are very natural and the strange wave-effect does not happen, thanks to the friction.

Robustness
Another aspect to consider in order to choose the right cloth type for the application is the robustness, or in simpler words: does the cloth explode resulting in unexpected behavior and crash of the application?

- **Positional:** the cloth with positional constraint is the most solid out of the cloths. It very rarely explodes, when it does it just shrinks very fast and is barely visible in the application.
- **Spring:** the cloth with spring constraints is the least robust, it happens very frequently that, when spawned on top of an object, it starts to produce spikes and, in a few seconds it shrinks and the application crashes.
- **Spring and Friction:** the cloth with spring and friction constraint is better than the cloth with spring constraint. It does not explode when spawned on top of another object but it is very sensitive to the value of K and U (elastic coefficient and damping coefficient), if they are over 13 and constraint iteration over 5, the cloth explodes enlarging and producing black stripes.

In conclusion, if in your application the cloth is just a secondary visual element, I will suggest to use the cloth with positional constraint since the framerate is high and stable, it is robust and even if it is that visually appealing, it will not heavily impact your application. On the other hand, if the cloth is an important element of your application I will suggest choosing the cloth with spring and

friction constraint: it will give a good visual effect, it is quite robust and, with a fine tuning it will not affect too much the efficiency of your application.

Next steps

One of the most obvious next steps to take is to switch the calculation and application of forces and constraints with compute shaders in order to make the simulation more efficient but it may cause different behavior depending on the GPU that we are using.

We could also exploit the positional constraint implemented here to simulate rigid bodies. Particles will be placed on the surface of the object, they will be connected with positional constraint, or spring and friction if we want some soft-bodies, and by modifying their mass we could have nice emerging behavior.