Introduction

This assignment will give you the chance to implement a simple cloth simulation. We will leverage our new found expertise on [finite element methods](#) to build a FEM cloth simulation. This simulation will use triangles, rather than tetrahedron as the finite elements and will use a principal stretch-based model for the cloth material. You will also implement your first contact response model, a simple velocity filter that can be bolted onto standard time integration schemes.

Prerequisite installation

On all platforms, we will assume you have installed cmake and a modern c++ compiler on Mac OS X[1], Linux[2], or Windows[3].

We also assume that you have cloned this repository using the `--recursive` flag (if not then issue `git submodule update --init --recursive`).

Note: We only officially support these assignments on Ubuntu Linux 18.04 (the OS the teaching labs are running) and OSX 10.13 (the OS I use on my personal laptop). While they *should* work on other operating systems, we make no guarantees.

All grading of assignments is done on Linux 18.04

Layout

All assignments will have a similar directory and file layout:

```
README.md
CMakeLists.txt
main.cpp
assignment_setup.h
include/
  function1.h
  function2.h
  ...
src/
  function1.cpp
  function2.cpp
  ...
data/
  ...
...
```

The `README.md` file will describe the background, contents and tasks of the assignment.

The `CMakeLists.txt` file setups up the cmake build routine for this assignment.

The `main.cpp` file will include the headers in the `include/` directory and link to the functions compiled in the `src/` directory. This file contains the `main` function that is executed when the program is run from the command line.

The `include/` directory contains one file for each function that you will implement as part of the assignment.

The `src/` directory contains *empty implementations* of the functions specified in the `include/` directory. This is where you will implement the parts of the assignment.

The `data/` directory contains *sample* input data for your program. Keep in mind you should create your own test data to verify your program as you write it. It is not necessarily sufficient that your program *only* works on the given sample data.

Compilation for Debugging

This and all following assignments will follow a typical cmake/make build routine. Starting in this directory, issue:

```
mkdir build
cd build
cmake ..
```

If you are using Mac or Linux, then issue:

```
make
```

Compilation for Testing

Compiling the code in the above manner will yield working, but very slow executables. To run the code at full speed, you should compile it in release mode. Starting in the **build directory**, do the following:

```
cmake .. -DCMAKE_BUILD_TYPE=Release
```

Followed by:

```
make
```

Your code should now run significantly (sometimes as much as ten times) faster.

If you are using Windows, then running `cmake ..` should have created a Visual Studio solution file called `a4-cloth-simulation.sln` that you can open and build from there. Building the project will generate an .exe file.

Why don't you try this right now?

Execution

Once built, you can execute the assignment from inside the `build/` using

```
./a4-cloth-simulation
```

While running, you can activate or de-activate the collision sphere by pressing `c`.

Background

In this assignment we will move from the simulation of volumetric objects to the simulation of thin sheets or thin shells. For thin objects, rather than discretize the volume with tetrahedra (as was done in [assignment 2](#) and [assignment 3](#)) we discretize the [medial surface](#) of the cloth. This surface takes the form of a two-dimensional (2d) [manifold](#) embedded in (3d) space. While many of the concepts you have already learned will carry over to this assignment, the major confounding factor will be evaluating the material model for the cloth on this manifold. To make things more interesting, we will see our first material model expressed in terms of the "principal stretches" of the deformation.

In order to allow for more interesting interactions with the cloth, we will also implement collision detection and resolution with an analytical sphere. We will implement a simple collision resolution scheme, via velocity filter. These algorithms try to prevent collisions by "filtering" a previously computed velocity to remove any components that might make collisions worse. While running the assignment code, you can activate or de-activate the collision sphere by pressing `c`.
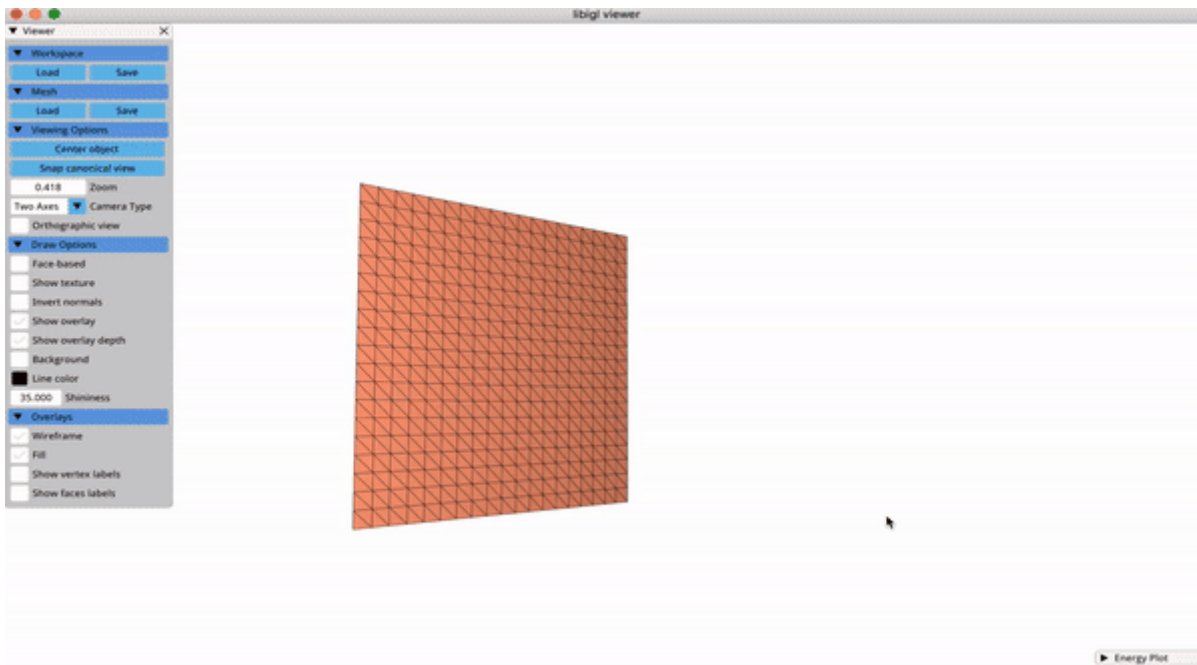
Cloth simulation!

Resources

Somewhat sadly, it is difficult to find a comprehensive resource for modern cloth simulation. However, there are a collection of seminal papers that are helpful to peruse. The first, [Large-Steps in Cloth Simulation](#) is widely recognized for introducing the linearly-implicit time integrator to graphics. While we won't deal with bending stiffness (think the difference between a Kleenex and a steel sheet), [Discrete quadratic curvature energies](#) are still the state-of-the-art approach for triangle mesh cloth. Finally, modern cloth simulators rely on [aggressive remeshing schemes](#) to capture detail while retaining performance.

Finite Elements on Manifolds

Our major challenge in this assignment arises from the difference in the dimensionality of the cloth material and the world (deformed) space (**NOTE:** I use the terms world and deformed space interchangeably). Cloth is locally two-dimensional (*2d*) while the world is [*3d*](#). What will be comforting is that, a relatively straight-forward application of the finite-element-method (FEM) will allow us to build a passable dynamic cloth simulator.

Triangular Finite Elements

The previous assignment applied FEM to *volumetric* simulation -- the simulation of objects with geometry of dimension equal to that of the world space (i.e our bunny and armadillo were 3d as was the world). In this case our finite elements were also volumetric ... they were tetrahedra in the undeformed space of the simulated object.

In the case of cloth, the underformed geometry is of different dimension (2d) than the world space (3d). Because our finite elements divide up the undeformed space, they also need to be 2d. As such we will use [triangles,](#) not tetrahedra as our elements.

Generalized Coordinates and Velocities

Just as in the [previous assignment](#), we need to choose basis, or shape functions with which to approximate functions on our, now triangular, mesh. A triangle as 3 nodes our approximations become

$$f\left(\mathbf{Y}\right) = \sum_{i=0}^{2} f_i \phi_i\left(\mathbf{Y}\right)$$

where ϕ_i are the [barycentric coordinates](#) for a 2D triangle and $\mathbf{Y} \in \mathcal{R}^2$ is the 2d coordinate in the undeformed space.

However, cloth is really a thin volumetric construct, of which our triangle only represents a small part. We might need information lying slightly off the triangle surface. To account for this we will need to modify our FEM model a bit. First, let's assume our triangle is actually embedded in a 3D undeformed space $\mathbf{X} \in \mathcal{R}^3$. Let's try and and build and appropriate mapping from this space to the world space.

Given any point $\mathbf{X}$ in the undeformed space, we can compute the barycentric coordinates of the nearest point on our triangle by solving

$$\begin{bmatrix} \phi_1\left(\mathbf{X}\right) \\ \phi_2\left(\mathbf{X}\right) \end{bmatrix} = \left(T^T T\right)^{-1} T^T \left(\mathbf{X} - \mathbf{X}_0\right)$$

,

where $T = \left[\,\left(\mathbf{X}_1 - \mathbf{X}_0\right) \quad \left(\mathbf{X}_2 - \mathbf{X}_0\right)\,\right]$ is a matrix of edge vectors. We use the constriaint $\phi_0 + \phi_1 + \phi_2 = 1$ to reconstruct ϕ_0. This equation finds the barycentric coordinates of the nearest point on the triangle to $\mathbf{X}$ in a least squares fashion. The error in this least squares solve will be orthogonal to the column space of T, our triangle. For any point $\mathbf{X}$ we can work out its offset from the triangle by computing $(\mathbf{X} - \mathbf{X}_0)^T \mathbf{N}$, where $\mathbf{X}_0$ is the first vertex of our triangle and $\mathbf{N}$ is the undeformed surface normal of the triangle. Because our triangle has a constant normal, we don't need to worry about where we compute it, which makes this all very convenient.

Let's assume that our point $\mathbf{X}$ maintains a constant offset from the triangle when deformed. This implies we can reconstruct the world space position by offsetting our point the same distance along the world space normal $\mathbf{n}$. This gives us the following mapping from reference space to world space

$$x\left(\mathbf{X}\right) = \sum_{i=0}^{2} \mathbf{x}_i \phi_i\left(\mathbf{X}\right) + \left(\mathbf{X} - \mathbf{X}_0\right)^T \mathbf{N} \cdot \mathbf{n}\left(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2\right)$$

.

Now we can choose the **generalized coordinates** ($\mathbf{q} \in \mathcal{R}^9$) to be the stacked vector of vertex positions, which defines the **generalized velocities** as the stacked *9d* vector of per-vertex velocities.

Deformation Gradient

There are lots of ways to handle build a cloth deformation gradient in [literature](#). In this assignment we will be able to avoid these more complicated solution due to our particular choice of undeformed to world space mapping which allows us to directly compute a 3×3 deformation gradient as

$$\mathbf{F} = \frac{\partial \mathbf{x}}{\partial \mathbf{X}} = \begin{pmatrix} \mathbf{x}_0 & \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{n} \end{pmatrix} \begin{pmatrix} -\mathbf{1}^T \left(\mathbf{T}^T \mathbf{T}\right)^{-1} \mathbf{T}^T \\ \left(\mathbf{T}^T \mathbf{T}\right)^{-1} \mathbf{T}^T \\ \mathbf{N}^T \end{pmatrix}$$

Kinetic Energy

Armed with the generalized velocities, the formula for the per-triangle kinetic energy is eerily similar to that of assignment 3. It's an integral of the local kinetic energy over the entire triangle, multiplied by the thickness of the cloth, h. For this assignment you are free to assume the thickness of the cloth is 1.

$$T_{triangle} = \frac{1}{2} \dot{\mathbf{q}}^T \left(h \int_\Gamma \rho \begin{pmatrix} \phi_0 \phi_0 \mathrm{I} & \phi_0 \phi_1 \mathrm{I} & \phi_0 \phi_2 \mathrm{I} \\ \phi_1 \phi_0 \mathrm{I} & \phi_1 \phi_1 \mathrm{I} & \phi_1 \phi_2 \mathrm{I} \\ \phi_2 \phi_0 \mathrm{I} & \phi_2 \phi_1 \mathrm{I} & \phi_2 \phi_2 \mathrm{I} \end{pmatrix} d\Gamma \right) \dot{\mathbf{q}}$$

and can be compute analytically using a symbolic math package. The per-element mass matrices for every cloth triangle can then be *assembled* into the mass matrix for the entire mesh.

Potential Energy

For this assignment we will use a different type of material model to describe the elastic behaviour of the cloth. This is motivated by the fact that cloth is typically very resistant to stretching. For these materials, a linear stress-strain relationship is often desirable. Unfortunately, cloth triangles also rotate a lot (every time they fold-over for instance). Rotations are **NOT** linear and so a purely linear relationship will suffer from severe artifacts. To avoid this we will build a material model that only measures the in plane deformation of the cloth via its *principal stretches*.

Principal Stretches

Recall that in the previous assignment we used the right Cauchy strain tensor ($F^T F$) to measure deformation and the rationale for using this was that it measures the squared deformed length of an arbitrary, infinitesimal line of material, $\mathbf{dX}$. In other words, $|\mathbf{dx}|^2 = \mathbf{dX}^T \left(F^T F \right) \mathbf{dX}$. Because F is symmetric and positive definite, we can perform an [eigendecomposition](#) such that $F^T F = V \Lambda V^T$ where V is the orthogonal matrix of eigenvectors and Λ is the diagonal matrix of eigenvalues. This means we can think of this squared length as $|\mathbf{dx}|^2 = \mathbf{d\hat{X}}^T \Lambda \mathbf{d\hat{X}}$ where $\mathbf{d\hat{X}} = U^T \mathbf{dX}$. In other words, if we transform $\mathbf{dX}$ just right, its deformation is completely characterized by Λ.

Λ are the eigenvalues of $F^T F$ and also the squared *[singular values](#)* of F. We call these singular values of F the [principal stretches](#). They measure deformation independently of the orientation (or rotation/reflection) of the finite element.

Linear Elasticity without the Pesky Rotations

Now we can formulate a linear elastic model using the principal stretches which "filters out" any rotational components. Much like the Neohookean material model, this model will have one energy term which measures deformation and one energy term that tries to preserve volume (well area in the case of cloth). We already know we can measure deformation using the principal stretches. We also know that the determinant of F measures the change in volume of a 3D object. In the volumetric case this determinant is just the product of the principal stretches.

$$\psi \left(s_0, s_1, s_2 \right) = \mu \sum_{i=0}^{2} \left(s_i - 1 \right)^2 + \frac{\lambda}{2} \left(s_0 + s_1 + s_2 - 3 \right)^2$$

where λ and μ are the material properties for the cloth. The first term in this model attempts to keep s_0 and s_1 close to one (limiting deformation) while the second term is attempting to preserve the volume of the deformed triangle (it's a linearization of the determinant). This model is called **co-rotational linear elasticity** because it is linear in the principal stretches but rotates *with* each finite element. When we use energy models to measure the in-plane stretching of the cloth (or membrane), we often refer to them as membrane energies.

The Gradient of Principal Stretch Models

The strain energy density for principal stretch models, like the one above, are relatively easy to implement and understand. This is a big reason we like them in graphics. We'll also see that the gradient of this model (needed for force computation) is also pretty easy to compute.

Really, the derivative we need to understand how to compute is $\frac{\partial \psi}{\partial F}$. Once we have this we can use $\frac{\partial F}{\partial \mathbf{q}}$ to compute the gradient wrt to the generalized coordinates. Conveniently, we have the following for principal stretch models.

$$\frac{\partial \psi}{\partial F} = U \underbrace{\begin{bmatrix} \frac{\partial \psi}{\partial s_0} & 0 & 0 \\ 0 & \frac{\partial \psi}{\partial s_1} & 0 \\ 0 & 0 & \frac{\partial \psi}{\partial s_2} \end{bmatrix}}_{dS} V^T$$

where $F = USV^T$ is the singular value decomposition.

The Hessian of Principal Stretch Models

Unfortunately, the gradient of the principal stretch energy is not enough. That's because our favourite implicit integrators require second order information to provide the stability and performance we crave in computer graphics. This is where things get messy. The good news is that, if we can just compute $\frac{\partial \psi}{\partial F \partial F}$ then we can use $\frac{\partial F}{\partial \mathbf{q}}$ to compute our Hessian wrt to the generalized coordinates (this follows from the linearity of the FEM formulation wrt to the generalized coordinates). This formula is going to get ugly so, in an attempt to make it somewhat clear, we are going to consider derivatives wrt to single entries of F, denoted F_{ij}. In this context we are trying to compute

$$\frac{\partial}{\partial F_{ij}} \frac{\partial \psi}{\partial F} = \frac{\partial U}{\partial F_{ij}} dS V^T + U \text{diag} \left(\mathbf{ds}_{ij} \right) V^T + U dS \frac{\partial V}{\partial F_{ij}}^T$$

Here diag $()$ takes a 3×1 vector as input and converts it into a diagonal matrix, with the entries of the matrix on the diagonal. In our case, we define $\mathbf{ds}$ as

$$\mathbf{ds}_{ij} = \begin{bmatrix} \frac{\partial^2 \psi}{\partial s_0^2} & \frac{\partial^2 \psi}{\partial s_0 \partial s_1} & \frac{\partial^2 \psi}{\partial s_0 \partial s_2} \\ \frac{\partial^2 \psi}{\partial s_0 \partial s_1} & \frac{\partial^2 \psi}{\partial s_1^2} & \frac{\partial^2 \psi}{\partial s_1 \partial s_2} \\ \frac{\partial^2 \psi}{\partial s_0 \partial s_2} & \frac{\partial^2 \psi}{\partial s_1 \partial s_1 2} & \frac{\partial^2 \psi}{\partial s_2^2} \end{bmatrix} \begin{bmatrix} \frac{\partial s_0}{\partial F_{ij}} \\ \frac{\partial s_1}{\partial F_{ij}} \\ \frac{\partial s_2}{\partial F_{ij}} \end{bmatrix}$$

The first thing to keep in mind when looking at this formula is **DON'T PANIC**. It's a straight forward application of the chain rule, just a little nastier than usual. The second thing to keep in mind is that all derivatives wrt to s_2 are zero (it never changes, it's always 0). The final thing to keep in mind is that **I am giving you the code to compute SVD derivatives in dsvd.h/dsvd.cpp**.

If we define the svd of a matrix as $F = USV^T$, this code returns $\frac{\partial U}{\partial F} \in \mathcal{R}^{3 \times 3 \times 3 \times 3}$, $\frac{\partial V}{\partial F} \in \mathcal{R}^{3 \times 3 \times 3 \times 3}$ and $\frac{\partial S}{\partial F} \in \mathcal{R}^{3 \times 3 \times 3}$. Yes this code returns 3 and four dimensional tensors storing this quantities, yes I said never to do this in class, consider this the exception that makes the rule. The latter two indices on each tensor are the i and j indices used in the formula above.

The hardest part of implementing this gradient correctly is handling the SVD terms. These gradients have a different form based on whether your F matrix is square or rectangular. This is one big reason that the 3×3 deformation gradient we use in this assignment is desirable. It allows one to use the same singular value decomposition code for volumetric and cloth models.

Collision Detection with Spheres

To make this assignment a little more visually interesting, you will implement simple collision detection and resolution with an analytical sphere. **Collision Detection** is the process of detecting contact between two or more objects in the scene and **Collision Resolution** is the process of modifying the motion of the object in response to these detected collisions.

For this assignment we will implement per-vertex collision detection with the sphere. This is as simple as detecting if the distance from the center of the sphere to any vertex in your mesh is less than the radius of the sphere. If you detect such a collision, you need to store an **index** to the colliding vertex, along with the outward facing **contact normal(n)**. In this case, the outward facing contact normal is the sphere normal at the point of contact.

Collision Resolution

The minimal goal of any collision resolution algorithm is to prevent collisions from getting worse locally. To do this we will implement a simple *velocity filter* approach. Velocity filters are so named because the "filter out" components of an objects velocity that will increase the severity of a collision. Given a vertex that is colliding with our sphere, the only way that the collision can get worse locally is if that vertex moves *into* the sphere. One way we can check if this is happening is to compute the projection of the vertex velocity onto the outward facing contact normal ($\mathbf{n}^T\dot{\mathbf{q}}_i$, i selects the ith contacting vertex). If this number is > 0 we are OK, the vertex is moving away from the sphere. If this number is < 0 we better do something.

The thing we will do is project out, or filter out, the component of the velocity moving in the negative, normal direction like so:

$$\dot{\mathbf{q}}_i^{\text{filtered}} = \dot{\mathbf{q}}_i - \mathbf{n}\mathbf{n}^T\dot{\mathbf{q}}_i$$

This "fixes" the collision. This approach to collision resolution is fast but for more complicated scenes it is fraught with peril. For instance it doesn't take into account how it is deforming the simulated object which can cause big headaches when objects are stiff or rigid. We'll see a cleaner mathematical approach to content in the final assignment of the course.

Assignment Implementation

Implementation Notes

In this assignment you will reuse your linearly implicit integrator to time step the dynamic system. Also, we will eschew implementing the strain energy density function, quadrature rule and potential energies separately. Instead functions for potential energy and its derivatives should directly compute the integrated values for a triangular element.

dphi_cloth_triangle_dX.cpp

Piecewise constant gradient matrix for linear shape functions. Row i of the returned matrix contains the gradient of the ith shape function.

T_cloth.cpp

The kinetic energy of the whole cost mesh.

dV_cloth_gravity_dq.cpp

Gradient of potential energy due to gravity

V_membrane_corotational.cpp

Potential energy for the cloth stretching force

dV_membrane_corotational_dq.cpp

Gradient of the cloth stretching energy.

d2V_membrane_corotational_dq2.cpp

Hessian matrix of the cloth stretching energy

V_spring_particle_particle.cpp

Use your code from the last assignment

dV_spring_particle_particle_dq.cpp

Use your code from the last assignment

mass_matrix_mesh.cpp

Assemble the full mass matrix for the entire tetrahedral mesh.

assemble_forces.cpp

Assemble the global force vector for the finite element mesh.

assemble_stiffness.cpp

Assemble the global stiffness matrix for the finite element mesh.

linearly_implicit_euler.h

Use your code from the last assignment

fixed_point_constraints.cpp

Use your code from the last assignment

collision_detection_cloth_sphere.cpp

Detect if any mesh vertex falls inside a sphere centered at (0,0,0.4) with radius 0.22

velocity_filter_cloth_sphere.cpp

Project out components of the per-vertex velocities which are in the **positive** direction of the contact normal

pick_nearest_vertices.cpp

Use your code from the last assignment