Wrinkle Simulation Based on Elastic Solid with Embedded Thin Shell

Jiong Chen

April 7, 2014

1 Introduction

2 Related Work

3 Method Overview

Fine features such as wrinkles on the embedded surface can be observed when the elements of underlying lattice are sufficiently fine, requiring expensive models with hundreds of thousands if not millions, which will directly and steeply increases the time cost. Things would go worse and even unacceptable when we try to pursue real-time simulation in such situation. Therefore, our project builds upon the embedded mesh approach whose central inspiration is separating the function of determining model shape and displaying surface details by embedding a dense surface into a coarse volume. There are alternatives to unite both systems with position constraint and we simply use the barycentric interpolation.

For elastic model simulation, $\cdots$
$\cdots$

We solve the shell deformation, modeled as an unconstraint optimization problem, by using sequential quadratic programming. $\cdots$
$\cdots$

Algorithm 1 Framework of Wrinkle Simulation

```
Input:
    coarse tetrahedral mesh $\mathbf{Q} = (\mathbf{v}_q, \mathbf{f}_q)$ represents inner elastic model;
1:  Extract the surface of $\mathbf{Q}$ and then subdivide it to get a dense shell $\mathbf{T} = (\mathbf{v}_t, \mathbf{f}_t)$ as undeformed configuration
2:  Build the interpolation matrix $\mathcal{B}$ via barycentric interpolation s.t. $\mathbf{v}_t = \mathcal{B}\mathbf{v}_q$;
3:  Init a elastic model simulator with mesh $\mathbf{Q}$
4:  Init a shell deformer with mesh $\mathbf{T}$
5:  for each time step do
6:      Simulate the elastic model updating mesh $\mathbf{Q}$
7:      Calculate current embedded mesh distribution by $\mathcal{B}\mathbf{v}_q$
8:      Deform the thin shell $\mathbf{T}$ under shape reference of $\mathcal{B}\mathbf{v}_q$
9:      Render the shell
10: end for
```

4 Basic Energy Operator

4.1 Elastic

(NULL)

4.2 Shell

We follow the seminal work [Grinspun et al. 2003] to model the thin shell. In this way, the shell is governed by nonlinear *membrane* and *flexural* energies. These energies all measure differences between the undeformed configuration $\bar{\Omega}$ and deformed configuration Ω. According to the author, the measurements defined on the model remain invariant under rigid body transformations of the undeformed configuration and/or deformed configuration, which ensures that internal forces conserve linear and angular momentum.

Let $\varphi : \bar{\Omega} \to \Omega$ be piecewise-affine deformation map from the undeformed to the deformed surface, mapping every geometry element of the undeformed mesh to the corresponding element of the deformed mesh. We formulate the shell energy by the sum of membrane and flexural energies,

$$W = W_M + k_B W_B \tag{1}$$

In this formula, W_M is the membrane energy while W_B represents the flexural energy, where k_B is the bending stiffness. In our implementation, we just omit the measurement of local change in area and simply put $W_M = k_L W_L$, where $W_L = \sum_e (1 - \frac{\|e\|}{\|\bar{e}\|})^2 \|\bar{e}\|$, a summation over (length of) edges measuring local change in length. As for bending energy W_B, it is an extrinsic measure of the difference between the shape operator evaluated on the deformed and undeformed surfaces. We express this difference as the *squared difference of mean curvature:*

$$[\mathrm{Tr}(\varphi^*\mathrm{S}) - \mathrm{Tr}(\bar{\mathrm{S}})]^2 = 4(H \circ \varphi - \bar{H})^2, \tag{2}$$

where $\bar{\mathrm{S}}$ and S are the shape operators evaluated over the undeformed and deformed configurations respectively; likewise $\bar{H}$ and H are the mean curvatures; $\varphi^* S$ is the pull-back of S onto $\bar{\Omega}$, and we use $\mathrm{Tr}(\varphi^* S) = \varphi^* \mathrm{Tr}(S) = \mathrm{Tr}(S) \circ \varphi = H \circ \varphi$ for a diffeopmprphism φ. Integrating (2) over the reference domain we find the continuous flexural energy :

$$\int_{\bar{\Omega}} 4(H \circ \varphi - \bar{H}) d\bar{A} \tag{3}$$

Discretizing the integral over the *piecewise linear mesh* that represents the shell, we express our *discrete flexural energy* as a summation over mesh edges,

$$W_B(\mathbf{x}) = \sum_e (\theta_e - \bar{\theta}_e)^2 \|\bar{e}\| / \bar{h}_e, \tag{4}$$

where θ_e and $\bar{\theta}_e$ are corresponding complements of the dihedral angel of e measured in the deformed and undeformed configuration respectively, and $\bar{h}_e$ is a third of the weighted average of heights of the two triangles incident to the edge e.

There are several alternatives to approximate the matrixs B in our SQP method. Hessian matrix $\nabla^2 f$ and Guass-Newton matrix could be used, or a mixed scheme of both.

4.3 Constraint with Local Support

During each frame, we expect that the surface shell could follow the shape of current embedded mesh i.e. $\mathcal{B}\mathbf{v}_q$ synchronously and meanwhile fine features like visible wrinkles could appear locally on the surface. In order to obtain such effect, we adopt the constraint proposed by [Rémillard et al. 2013] and simply convert the linear equality constraint to a quadratic penalty energy. Then we could formulate our deformation as simply solving such an unconstrained optimization problem.

Position constraint is a indispensable bridge coupling surface and interior, thus I feel like to giving a specific statement of constructing the position constraint. The simplest scheme striking us at once is to set some fixed points, i.e. let $x_i = v_i$, for some i, just like we are holding on some points on the mesh. If we could generalize the concept of *single point* and regard so-called *single point* as an average of a cluster of points, then for a given cluster of vertices C, we will get position constraint in context with form

$$\sum_{i \in C} \alpha_i \mathbf{x}_i = \sum_{i \in C} \alpha_i [\mathcal{B}\mathbf{v}_q]_i \tag{5}$$

where the weights α_i provide an affine combination of the shell vertex positions $\mathbf{x}_i$ and the embedded mesh vertex positions $[\mathcal{B}\mathbf{v}_q]_i$. The weight is calculated as follow,

$$\alpha_i = \omega_C e^{-\frac{d_i^2}{2\sigma^2}},\ if\ d_i < 2\sigma,\ 0\ otherwise, \tag{6}$$

where ω_C normalizes the sum of weights to one, and d_i is the distance from vertex i to the cluster's center, which we approximate by the shortest path following mesh edges. [Rémillard et al. 2013] gives a clear algorithm description on how to generate a certain number of overlapping clusters centered at locations distributed evenly across the surface, which using breadth first search technique based on greedy expansion. Related implementation details are just omitted here.

Suppose we have got c clusters after above partition, we are able to define a constraint on each cluster, and together they form the rows of a spare constraint matrix H providing a linear transformation $H : \mathbb{R}^n \to \mathbb{R}^c$, with $c \ll n$. Now we could write our constraint as

$$H(\mathbf{x} - \mathcal{B}\mathbf{v}_q) = \mathbf{0}. \tag{7}$$

The constraint forces shell $\mathbf{x}$ to match the embedded shape $\mathcal{B}\mathbf{v}_q$, with the null space of H containing necessary deformations to produce wrinkles. We treat the shell deformation as isometric transformation thus H is a constant to time variation, so we needn't to update H within each time step. Considering the tradeoff between computing cost and visual effect, we select to convert (7) to a quadratic energy function, namely

$$W_P(\mathbf{x}) = k_P \|H(\mathbf{x} - \mathcal{B}\mathbf{v}_q)\|^2 \tag{8}$$

where k_P is weight need setting relatively large compared with k_B and k_L. Up till now, a well implemented unconstrained optimizer can be applied.

5 Acceleration via Cluster

Abundance of visual effect always stands on the opposite to the quality of interaction. With fast increasing number of freedom aiming to display more details, speed gradually becomes unacceptable for approaching real-time performance. Naturally, we choose to deploy our project on clusters to run it in a concurrent environment for acceleration.

It is obvious that we have to separate our model into several parts and send each of them to a corresponding node. We surprisingly find that such partition had been done exactly after we did a pre-processing on surface shell, using the greedy algorithm mentioned in section 4.3, to add position constraints. However, the generated regions are just sets of points and the shell energy functions to be optimized are all defined on triangle mesh though. To state conveniently, we treat the surface as a connected graph $G = (V, E)$ embedded in 3D space, where V is the set of vertices and E is the set of edges. Then what we have to do is to construct a sequence of subgraph $G_i = (V_i, E_i),\ for\ i = 1, 2, \cdots, n$ that $V_i \subseteq V, E_i \subseteq E$. In each subgraph, we call a point an *boundary vertex* if it lies on the convex hull when embedding such a patch onto a plane. And those other than boundary vertices are *inner vertex*. We use V_i^* do denote the inner vertex set of G_i, and ∂V_i for boundary vertex set. Here we have some properties should be satisfied among those subgraphs respect to our separation strategy:

1. $\bigcup_{i=1}^{n} G_i = G.$
2. $\forall G_i \subset G,\ E_i = \{(u, v) | u \in V_i \wedge v \in V_i \wedge (u, v) \in E\}.$
3. $\forall G_i, G_j \subset G,\ i \neq j \Rightarrow V_i^* \cap V_j^* = \emptyset.$
4. $\forall v \in \partial V_i \subseteq V_i,\ \bigvee_{j=1}^{n} v \in V_j^* = true.$

Condition 1 and 4 are sufficient to ensure that there is no omissive geometry elements that wouldn't be updated when delivering the optimization to several cluster's nodes, while condition 3 tries to minimize the cover of G formed through the union of all subgraphs. And the condition 2 ensures that every patch is a complete triangle mesh since all patches are induced subgraphs of the origin.

We use a simple approach to generate the separated triangle mesh satisfying the four properties above. Before that, it is necessary to record one-ring faces of every vertex in advance. Once the partition algorithm mentioned in section 4.3 being applied, we will get a number of disjoint point sets, based on which a patch could be constructed with ease just by collecting all vertices' one-ring faces within that cluster. Notice that some additional points will appear in a cluster after such procedure been carried out and it's not hard to find they share the name of boundary vertex. We likely have to merge some patches into a larger one according to the limited number of calculating nodes.

All concurrent program need to consider the problem of the critical region carefully. Our project is not beyond exception either. From above statement, some additional vertices will be added to each region and apparently the new ones belong to another different region initially. The coordinates of such vertices will be calculated in two nodes respectively and the values are different in all probability. From the condition 3 and 4, we can conclude that such vertex must be inside one and only that one cluster, and also a boundary vertex of another. To make it as simple as possible, we avoid the conflicts by just reserving the value contributed by the vertex serving as an inner one. (briefly claim some reasons).

We use the CS artitechure to build our parallel system.

6 Implementation Details

6.1 Interpolation

6.2 SQP

6.3 Modulize

7 Experiments

Coupling rigid with elastic solid. We use the *Bullet* together with our elastic solid simulator to generate a scene containing a ball falling free onto a sofa.

8 Discussion and TODO

deploy the project on clusters to get real-time simulation.

what is the essence of the difference between solving a problem by whole and seperating it into several parts.

9 Conclusion

10 Reference