CS 184: Computer Graphics and Imaging, Spring 2017

Project 4: Cloth Simulator

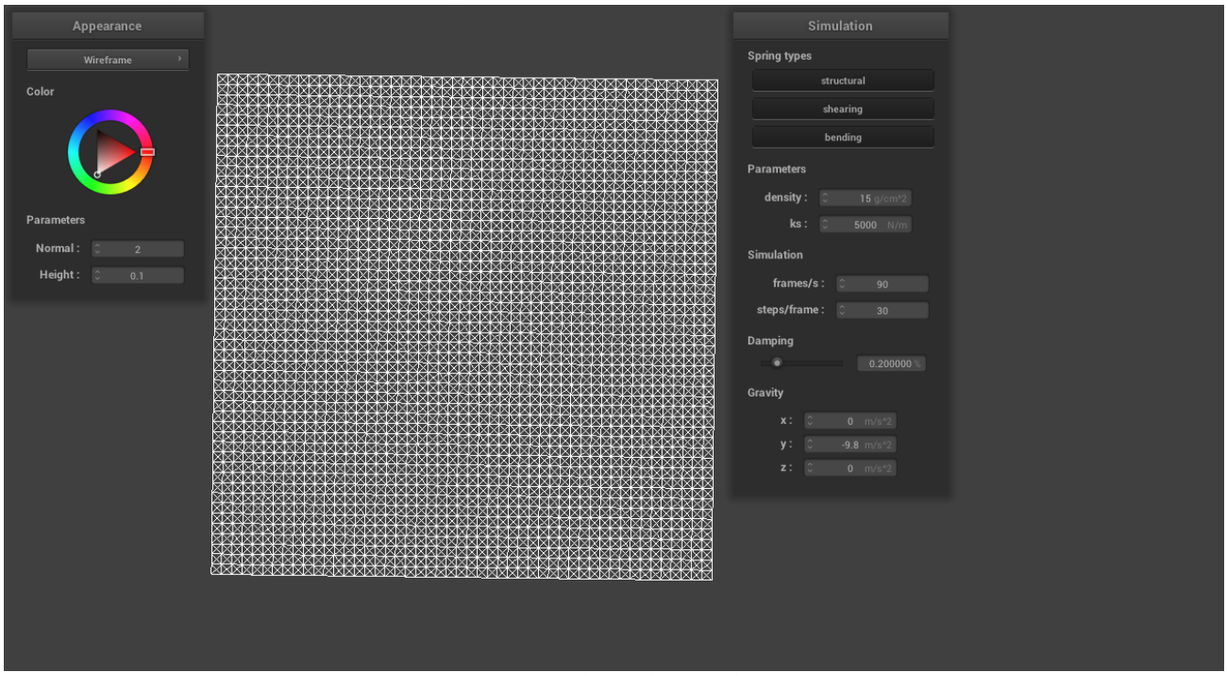
Shenao Zhang

Overview

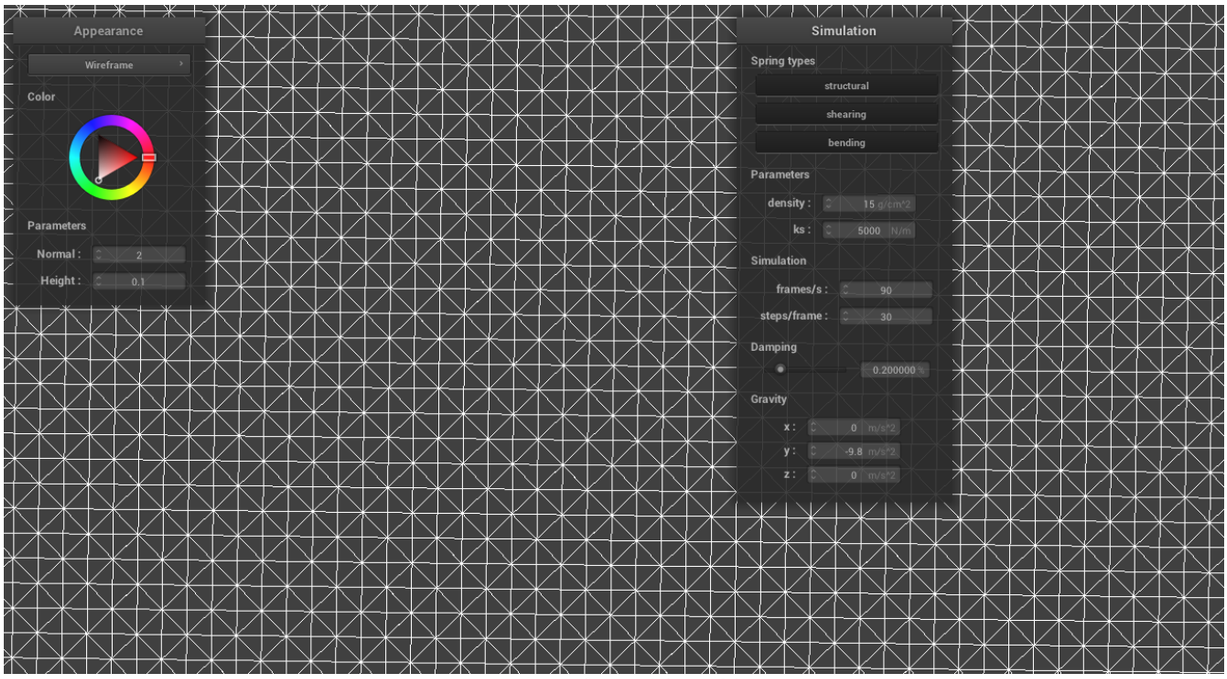
In this project, we will simulate the real world using physical knowledge, like simulate the forces caused by spring, gravity mainly using Newton's law. Then to better simulate the real world, we handle the cases that the cloth collisions with other objects and self-collisions. Then we can get the movement in these cases correct.Then we want to create different lighting and material effects. We implement different shaders to get the different material.

Part 1: Masses and springs

To simulate the colth, we can divide the cloth up into evenly spaced point masses and then connect nearby masses with springs. In this part, we will build a grid of masses and springs.
To create an evenly spaced grid of masses, we have two types of the cloth's orientation: HORIZONTAL and VERTICAL. For different orientations, we generate different point masses.
Then create springs to apply the structual, shea, and bending constraints between point masses.
After implementing these, we can get the following results.

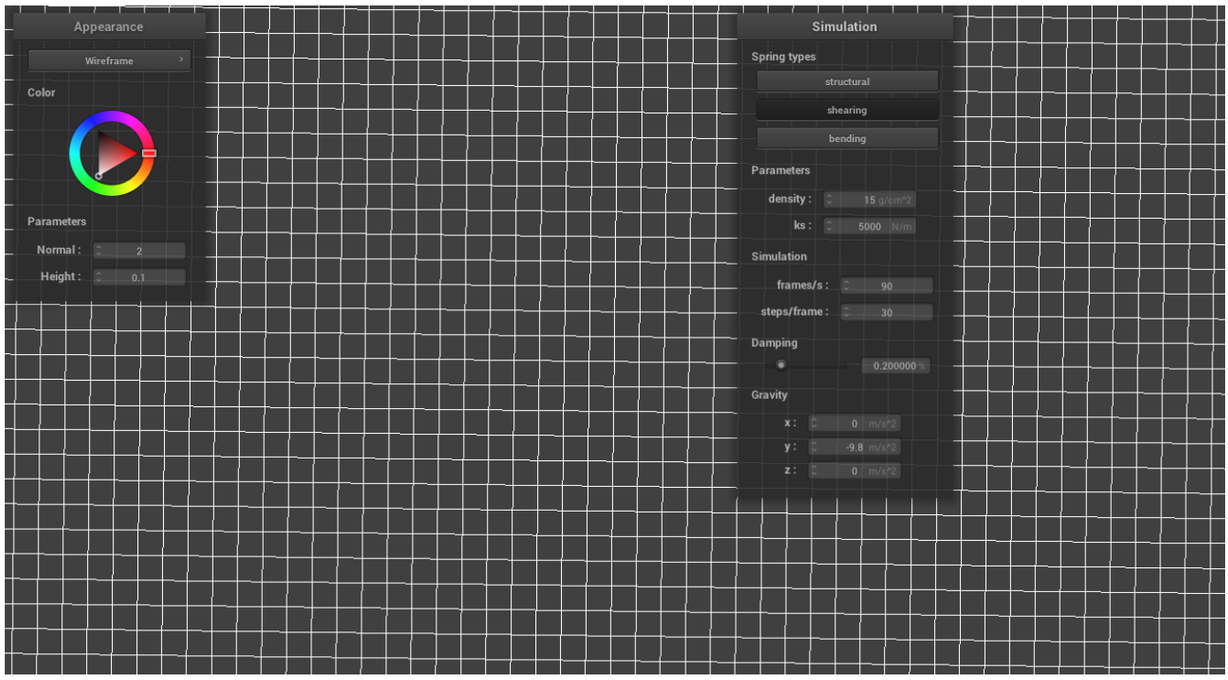


screenshots of scene/pinned2.json

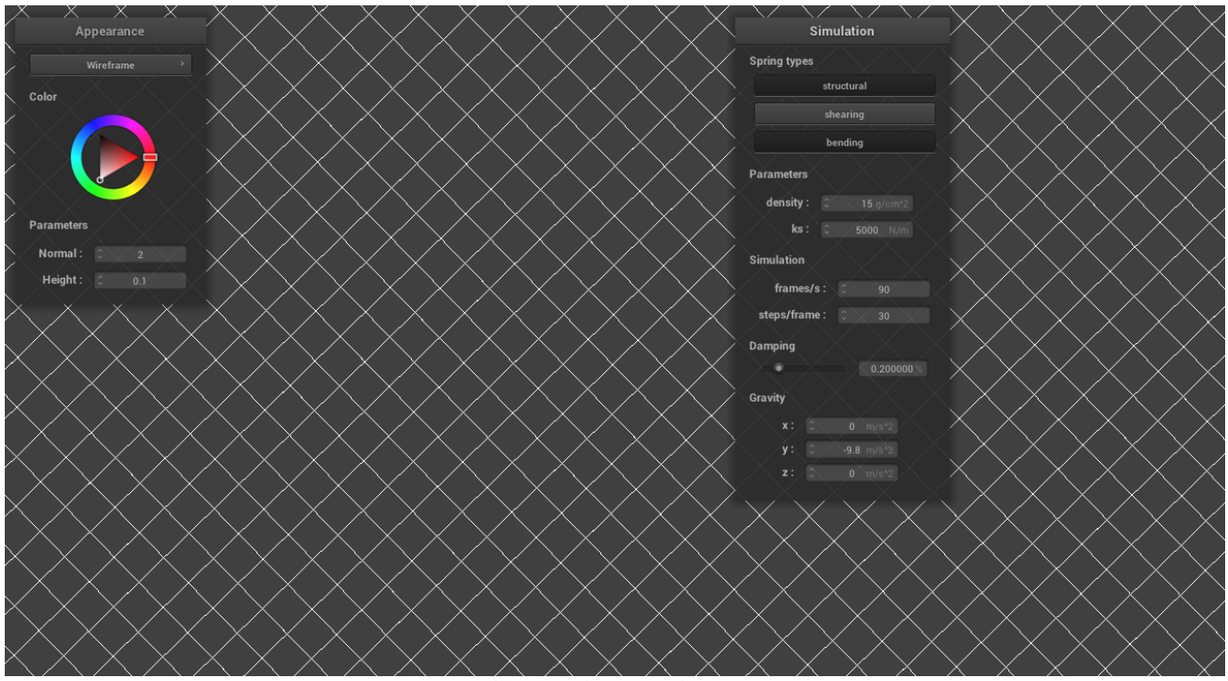


screenshots of scene/pinned2.json

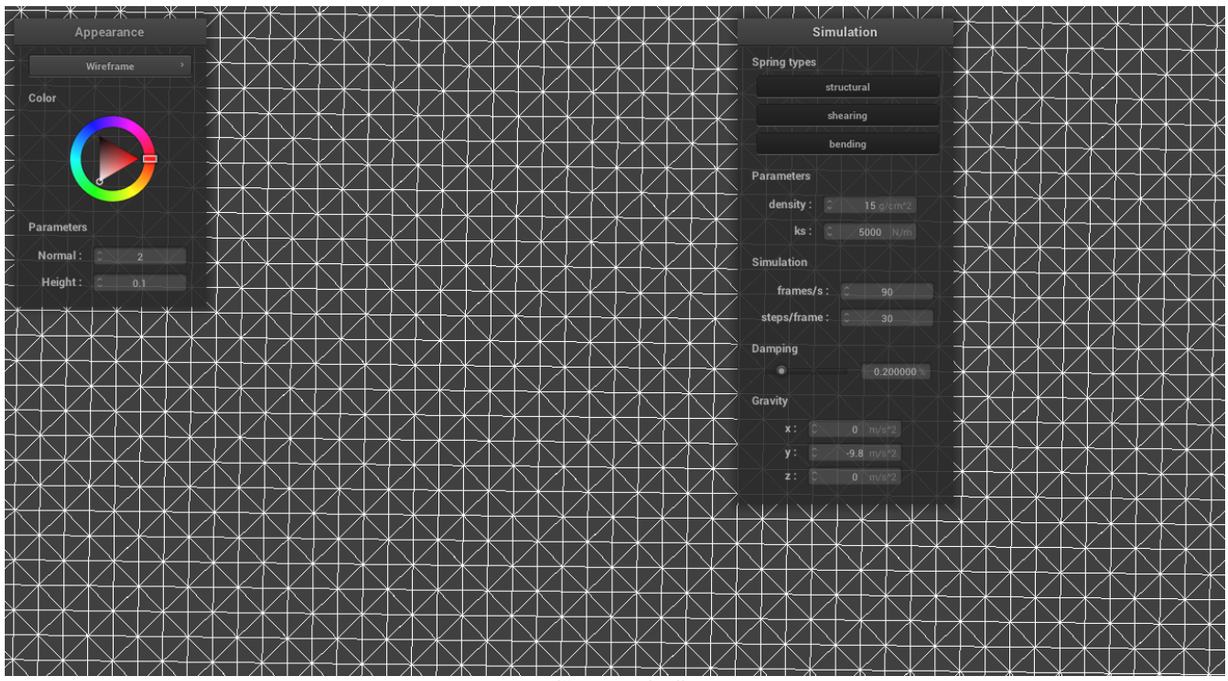
Now let me show what the wireframe looks like with different constraints.



scene/pinned2.json with only shearing constraints



scene/pinned2.json without any shearing constraints



scene/pinned2.json with all constraints

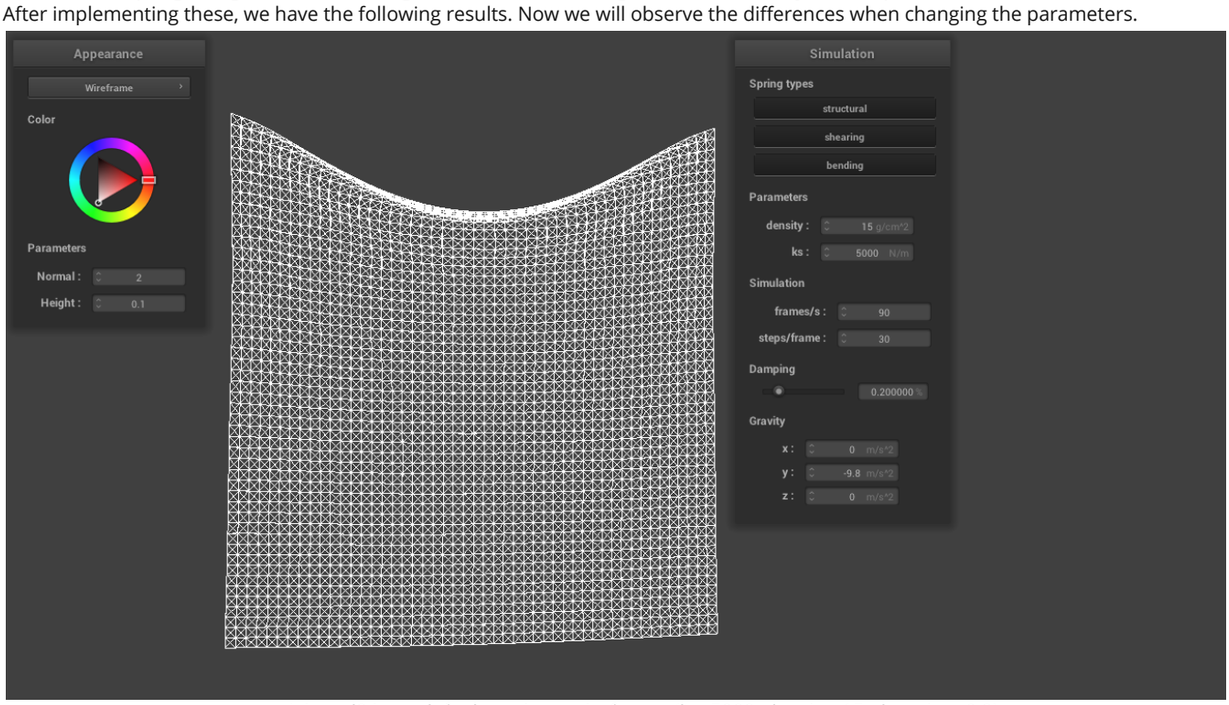
Part 2: Simulation via numerical integration

In the last part, we have set up the cloth model as a system of masses and springs, in this part, what we mainly do is to simulate via numerical integration according to the physical equations of motion in order to apply the forces on our cloth's point masses to figure out how they move from one time step to the next.
For each point mass, we have external forces (such as gravity) which uniformly affect the cloth. For each spring, we have spring

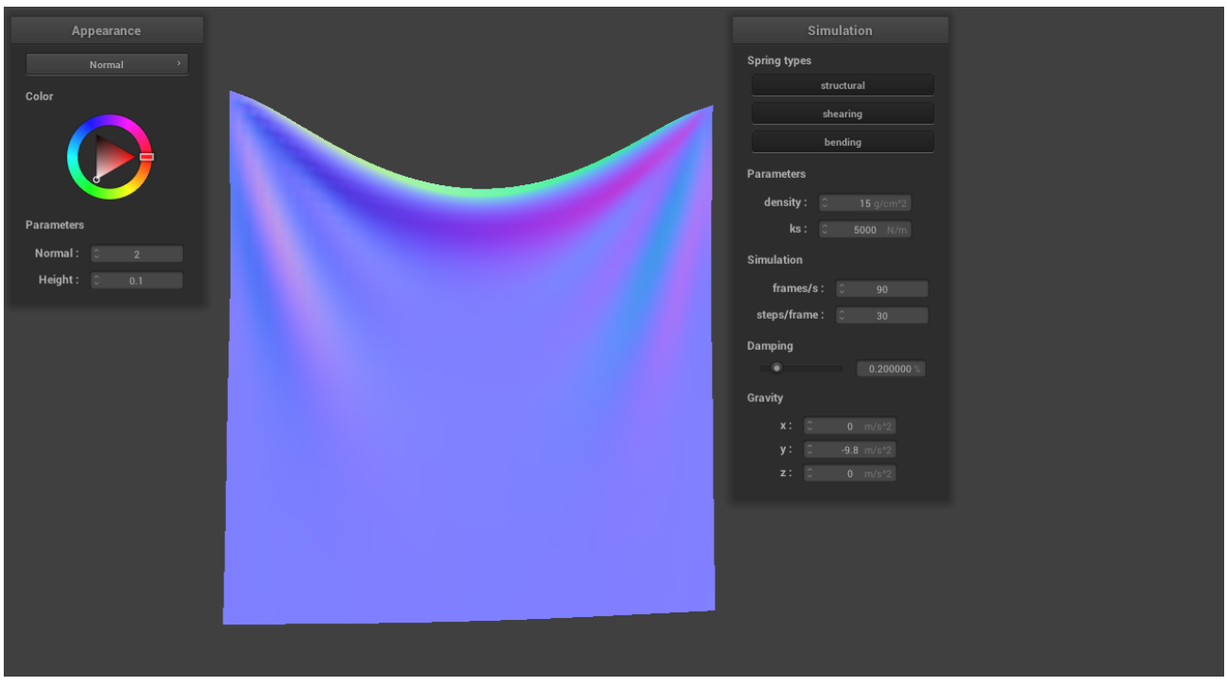
correction forces which apply the spring constraints from before to keep the cloth together.
Then we use Verlet integration as follows to compute new point mass positions.

$$x_{t+dt} = x_t + (1-d) * (x_t - x_{t-dt}) + a_t * dt^2$$

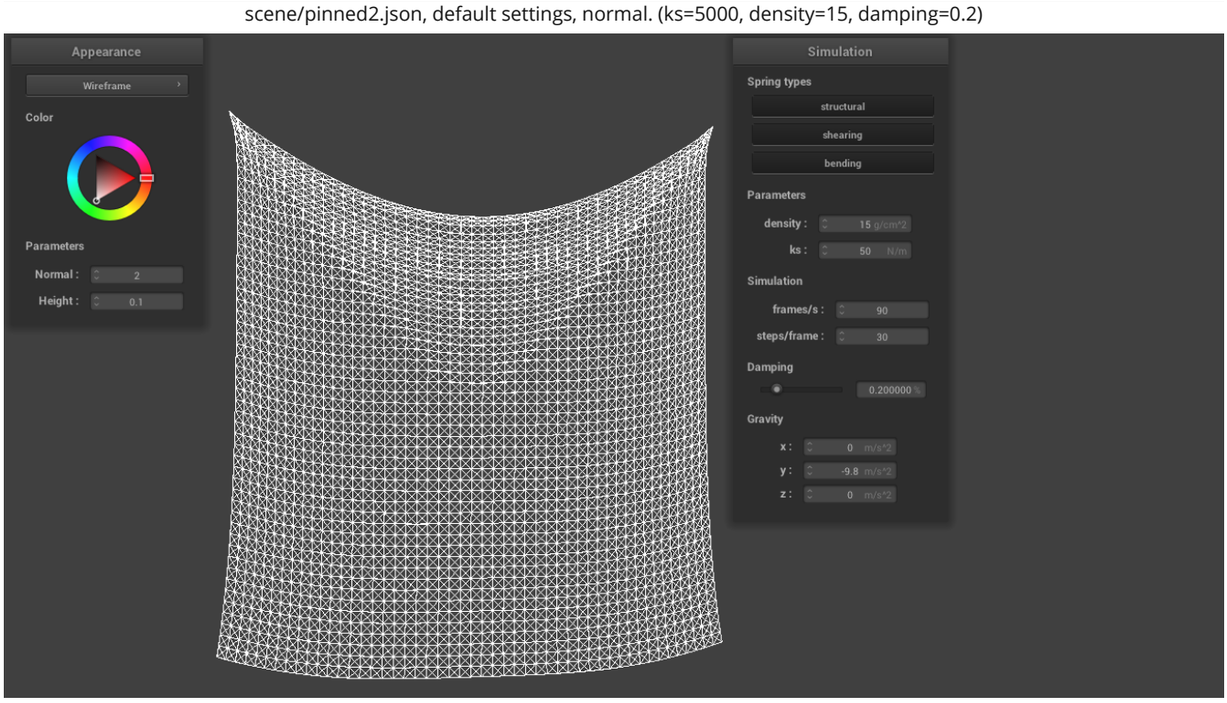
Then, to help keep springs from being unreasonably deformed during each time step, we will implement an additional feature on deformation constraints in mass-spring models. For each spring, apply this constraint by correcting the two point masses' positions such that the spring's length is at most 10% greater than its rest_length at the end of any time step.
After implementing these, we have the following results. Now we will observe the differences when changing the parameters.



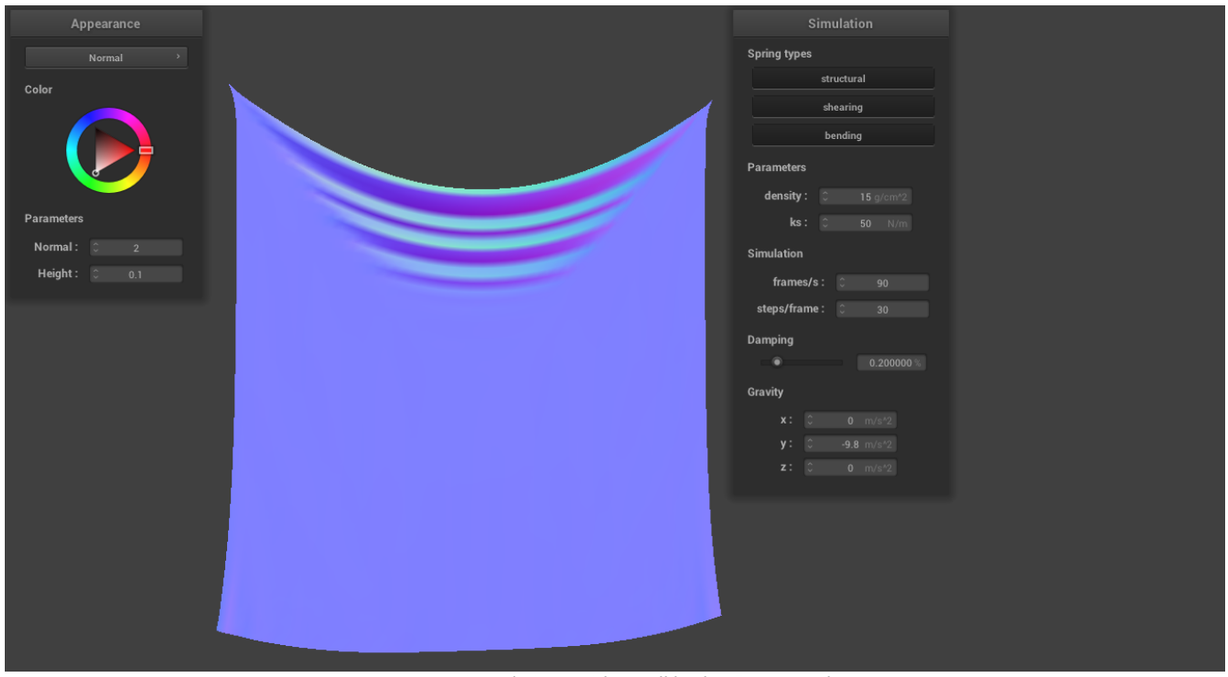
scene/pinned2.json, default settings, wireframe. (ks=5000, density=15, damping=0.2)



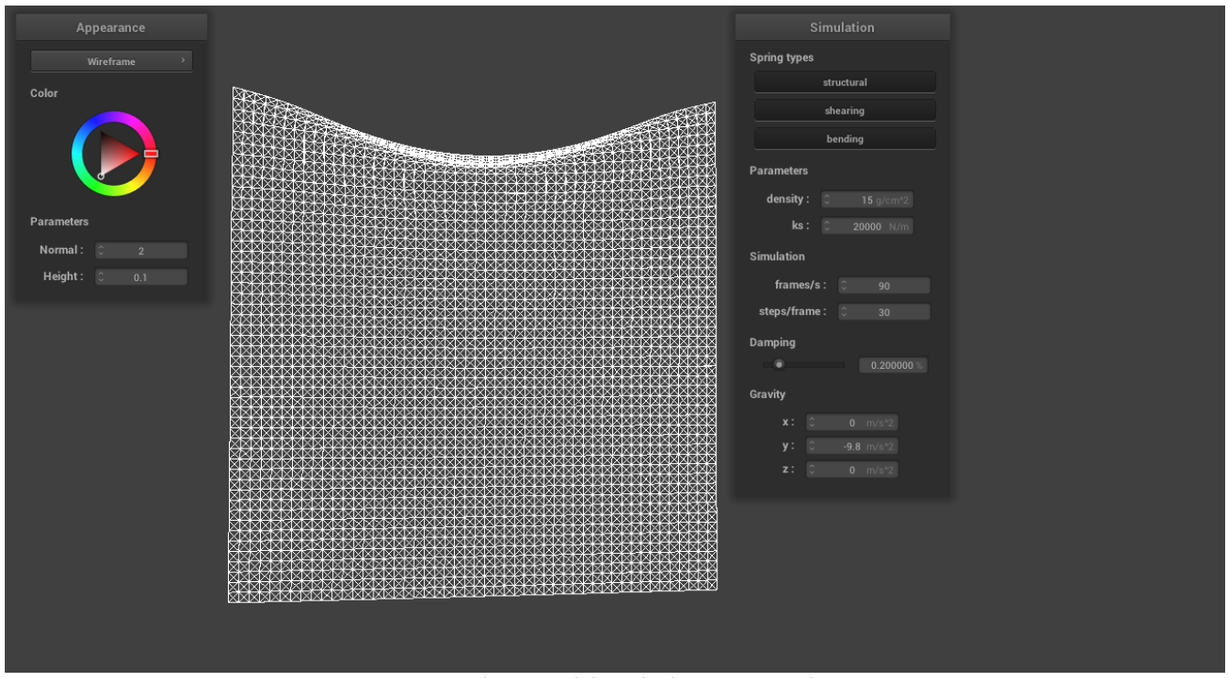
scene/pinned2.json, default settings, normal. (ks=5000, density=15, damping=0.2)



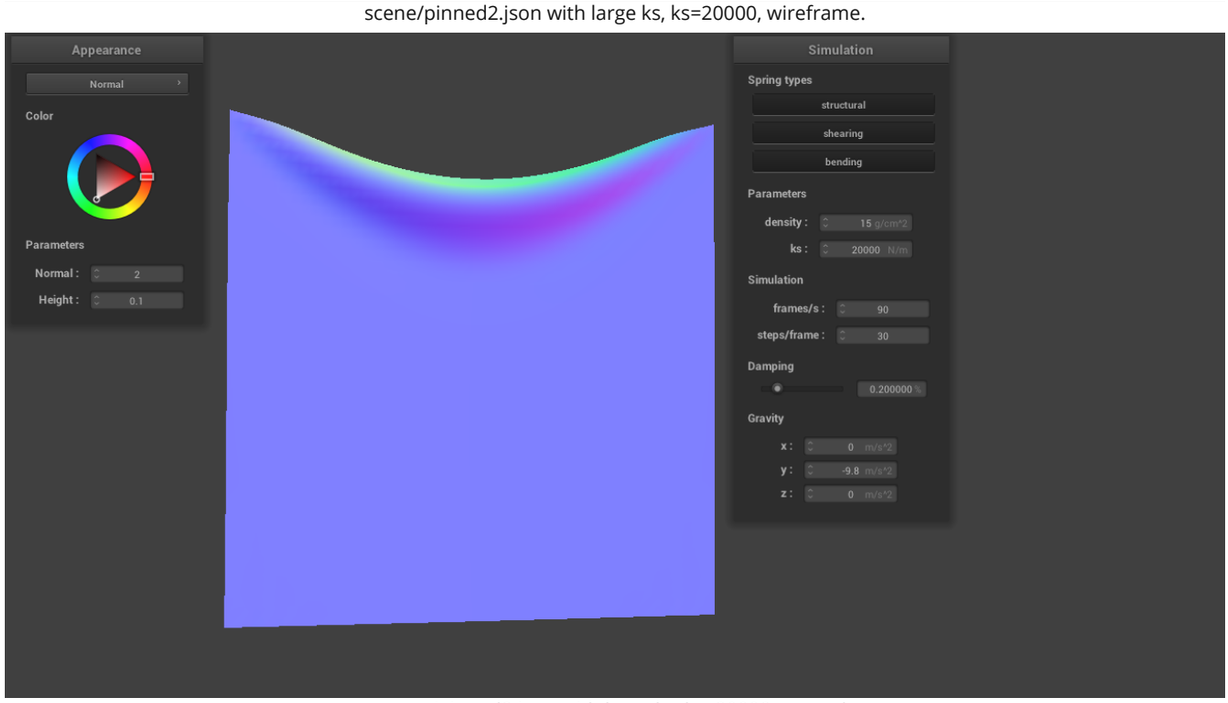
scene/pinned2.json with small ks, ks=50, wireframe.



scene/pinned2.json with small ks, ks=50, normal.

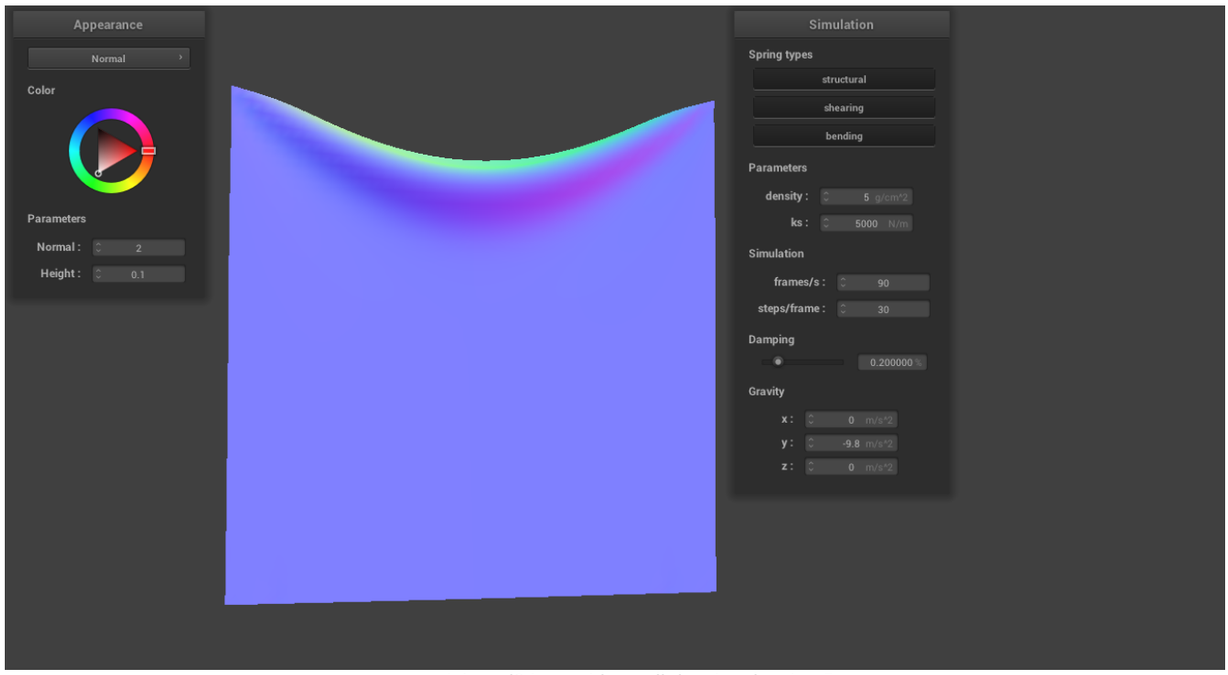


scene/pinned2.json with large ks, ks=20000, wireframe.

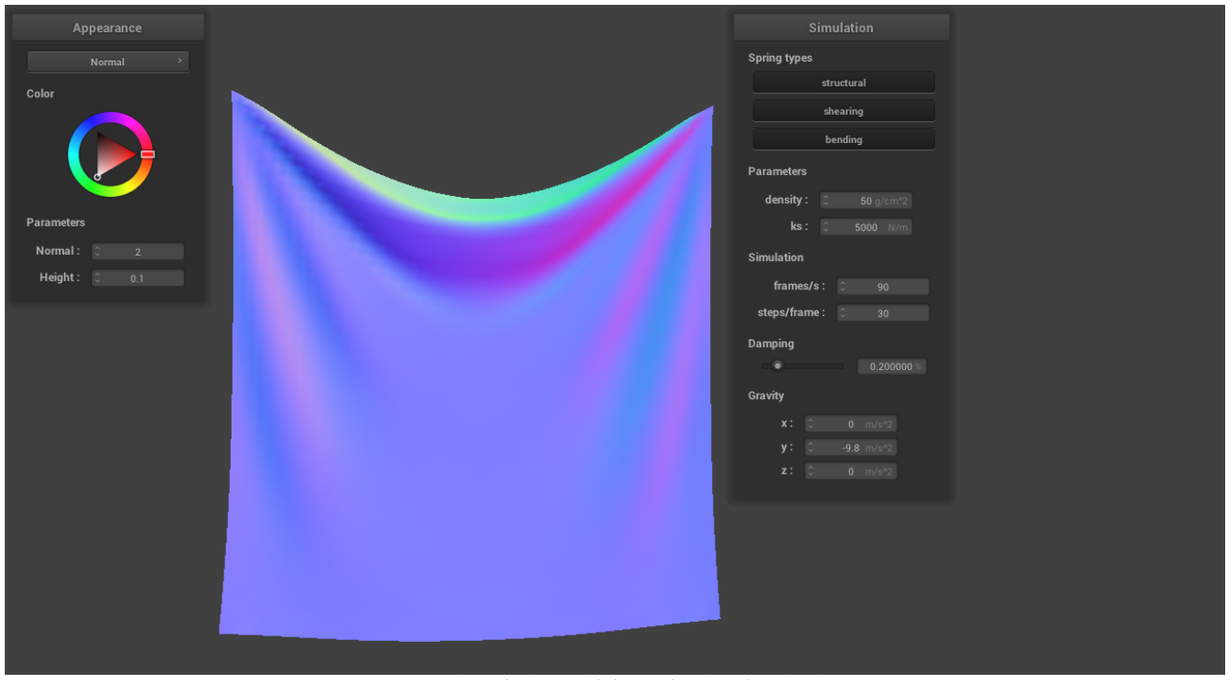


scene/pinned2.json with large ks, ks=20000, normal.

As we can see, with smaller ks, the cloth is more stretched, especially for the pinned top of the cloth; with larger ks, the cloth is more flat. This is because we have the same external force, which is caused by gravity, so to balance this external force, with smaller ks, we need to stretch the springs more to get balance. With larger ks, we don't need to stretch the springs that much, which causes that the cloth lokes flat.

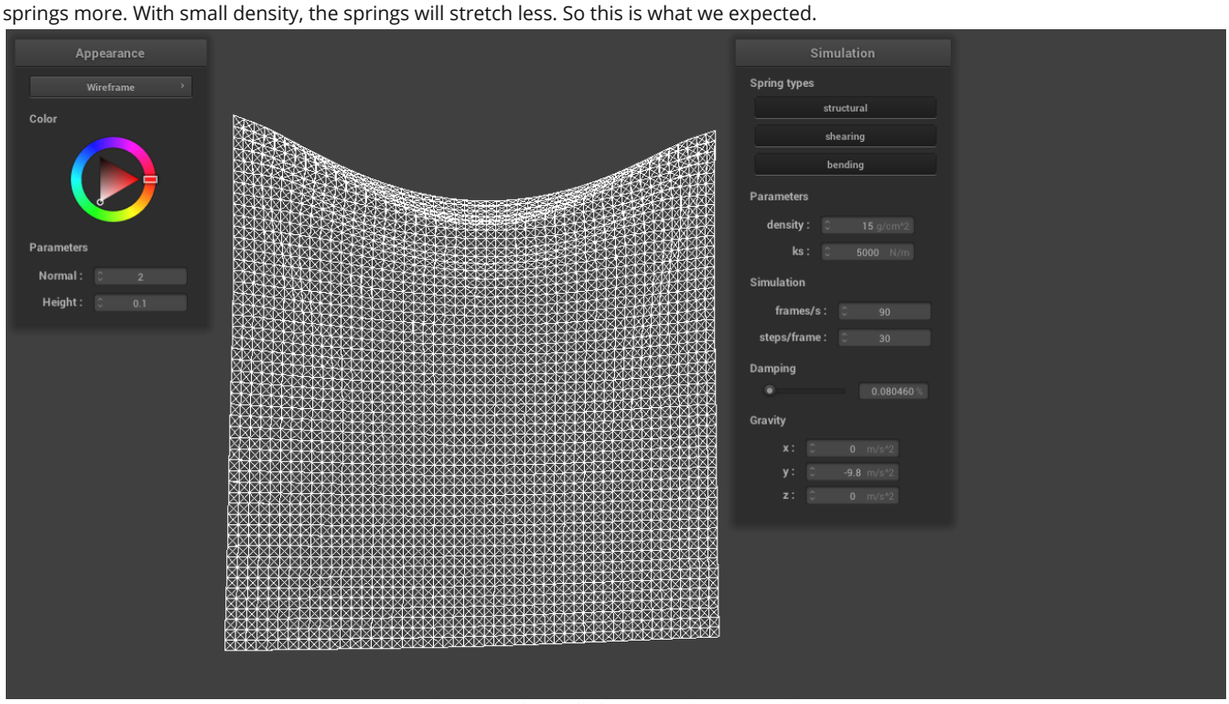


scene/pinned2.json with small density, density=5.

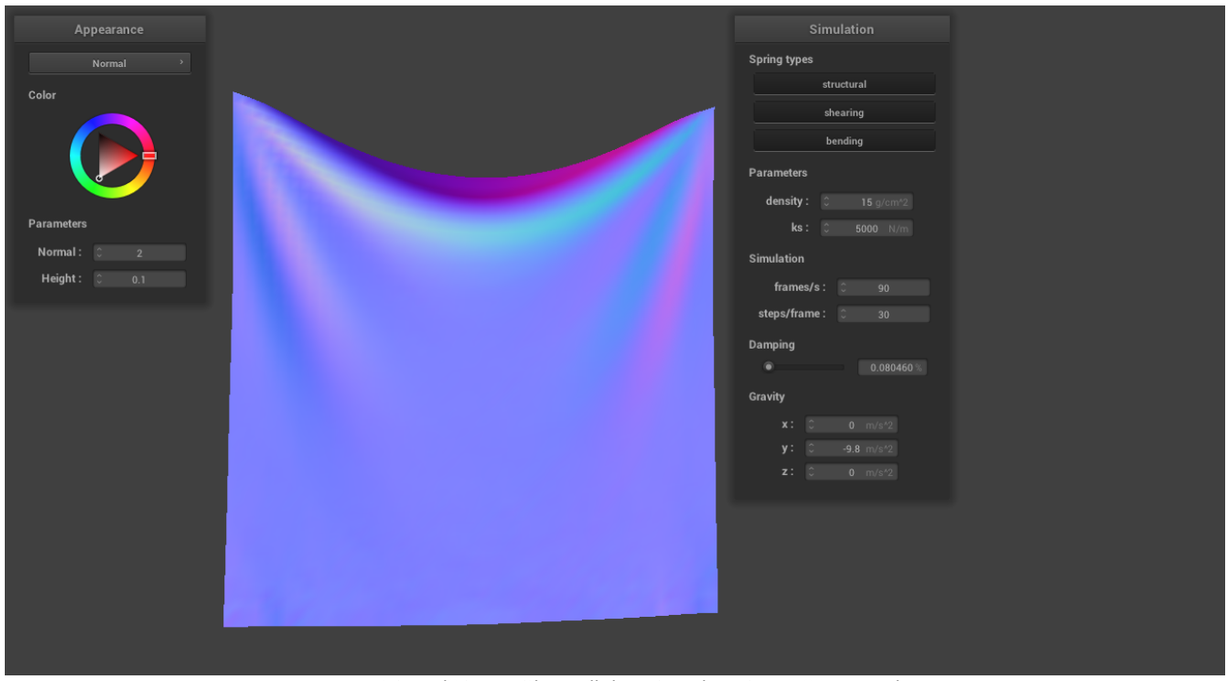


scene/pinned2.json with large density, density=50.

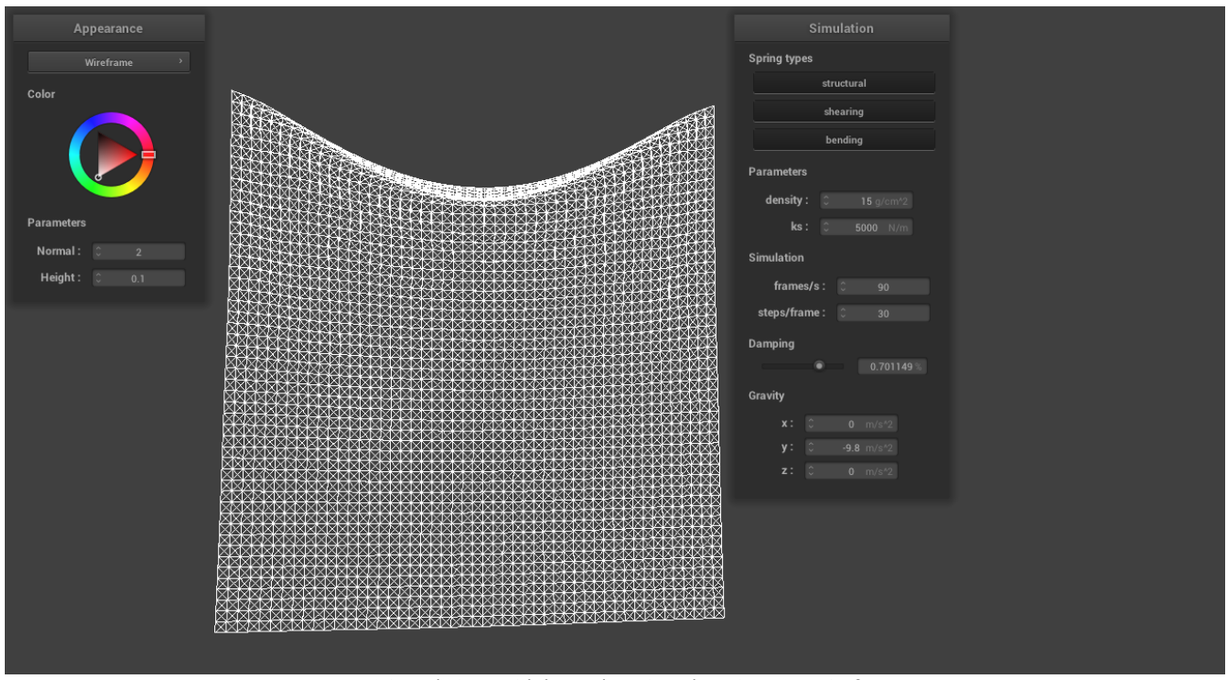
As we can see, with smaller density, the cloth looks flat, while with larger density, the cloth is more stretched at the top of the cloth. Since when the density is large, the external forces caused by gravity increase accordingly. To balance this force, we need stretch the springs more. With small density, the springs will stretch less. So this is what we expected.



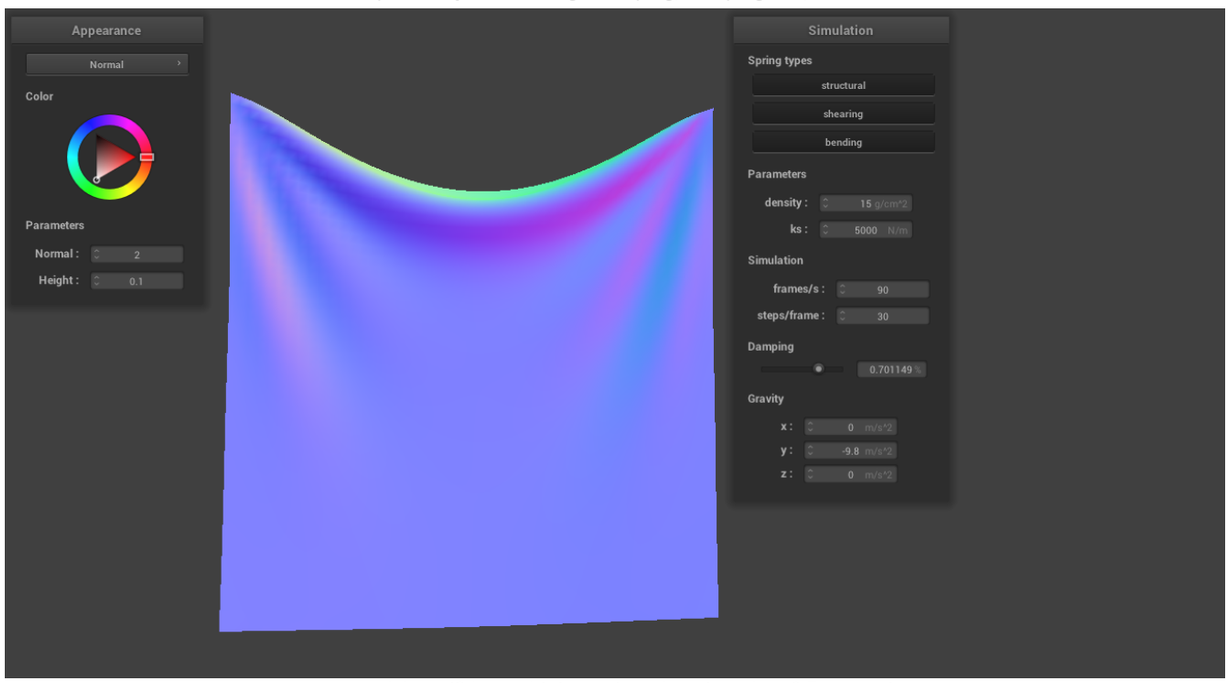
scene/pinned2.json with small damping, damping=0.08, wireframe.



scene/pinned2.json with small damping, damping=0.08, normal.



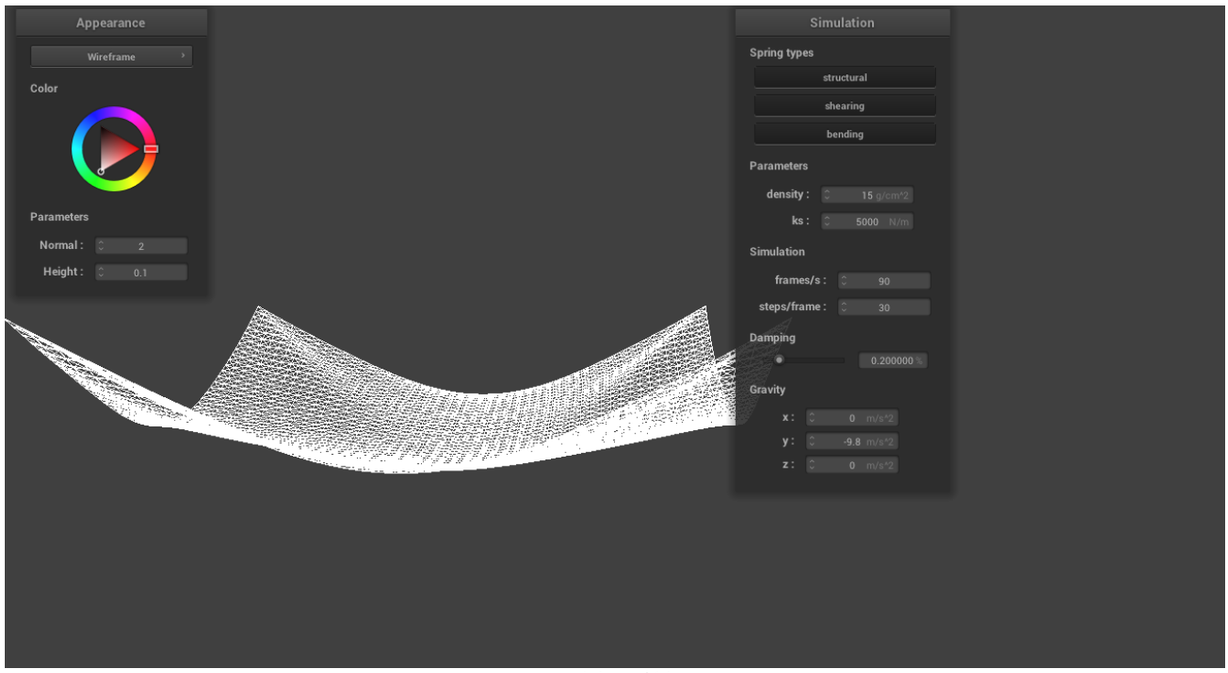
scene/pinned2.json with large damping, damping=0.7, wireframe.



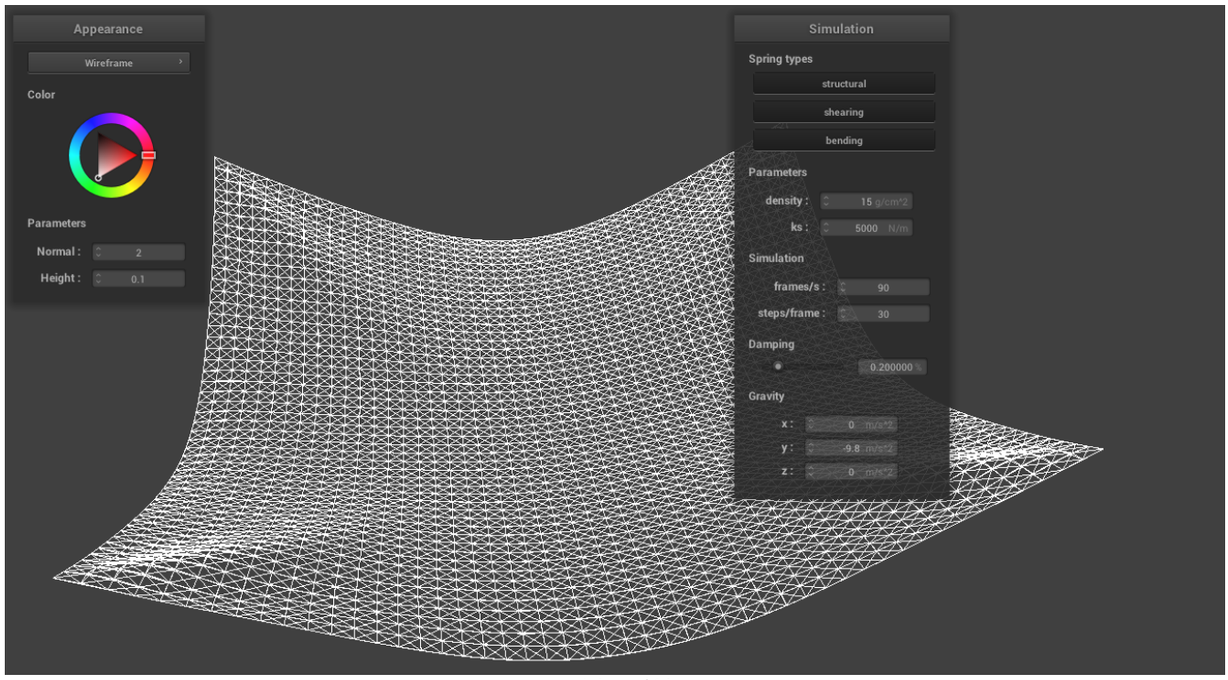
scene/pinned2.json with large damping, damping=0.7, normal.

The difference between different damping is that with larger damping, the cloth takes less time to move and stay oscillated, with smaller damping, it will take more time to get to balance. This is because when damping is big, the energy will lose quickly, thus it will take a short time to get to balance. i.e. less time to move and stay oscillated.

Then let me show what it looks like when pin the four corners. The results are as follows:



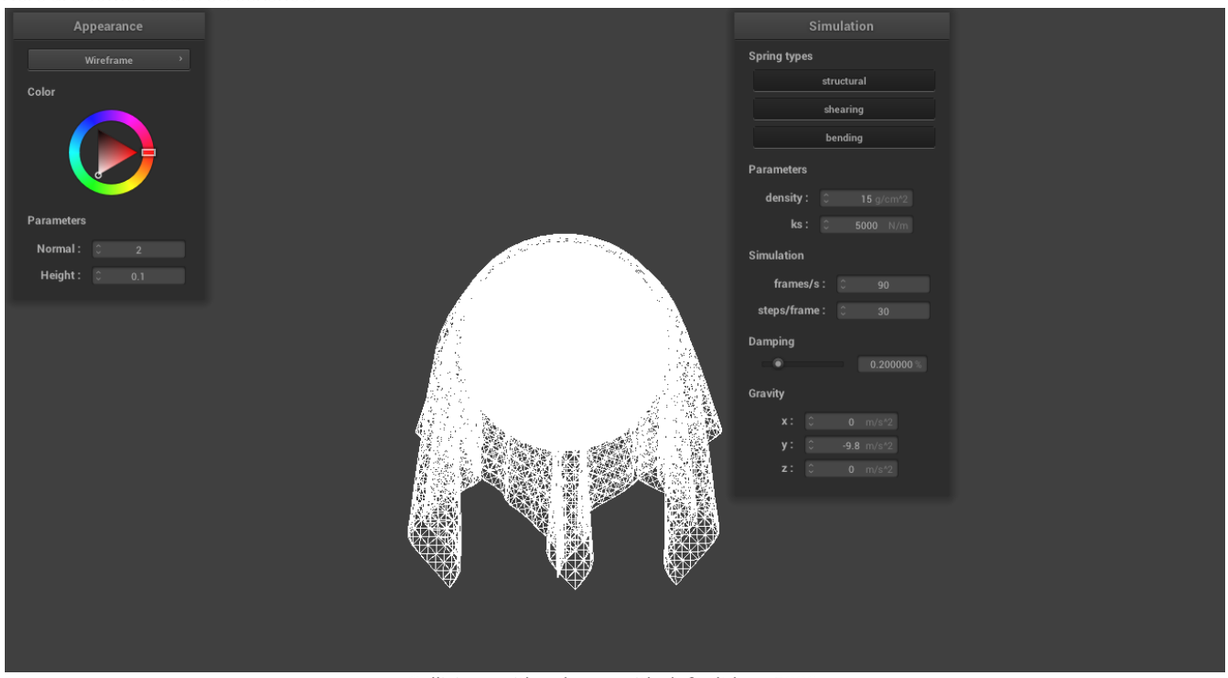
scene/pinned4.json



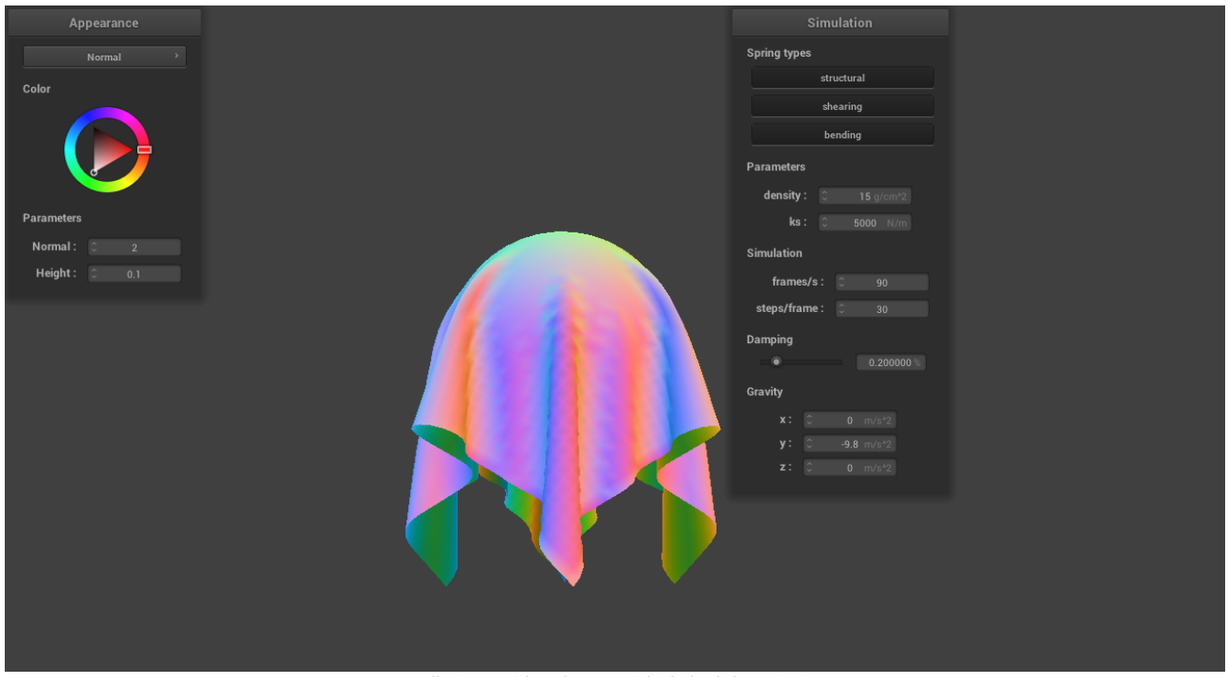
scene/pinned4.json

Part 3: Handling collisions with other objects

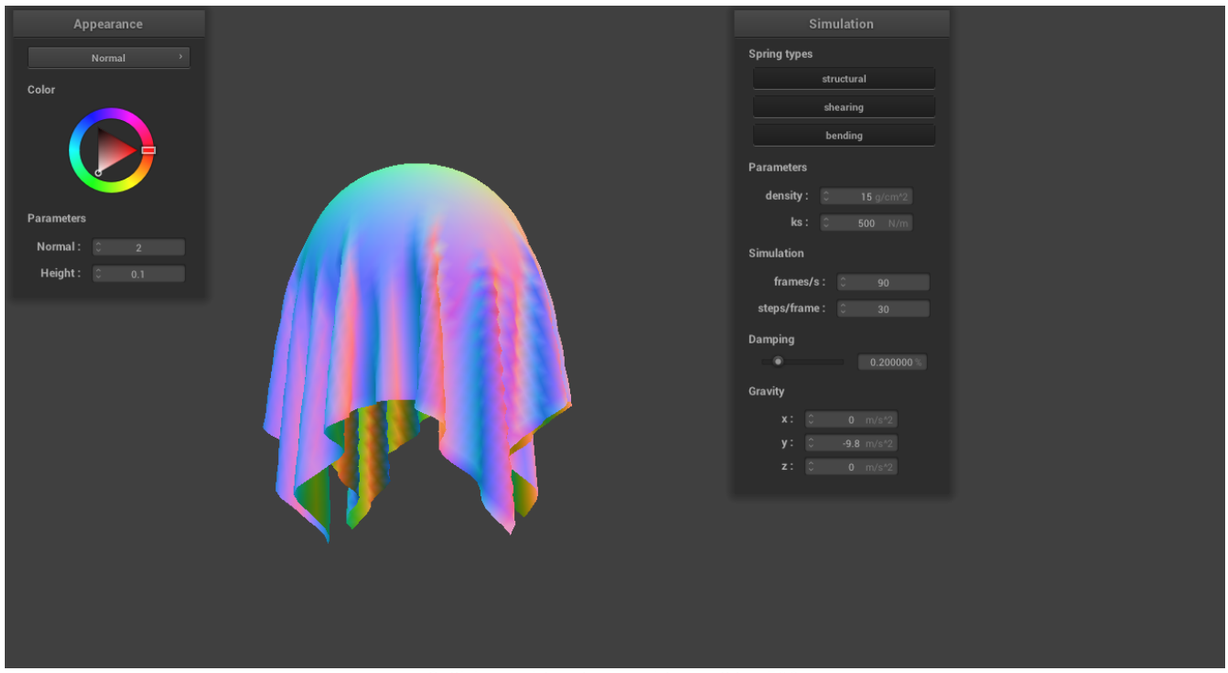
In this part, we add support for cloth collision with other objects in the scene to make things more interesting.
First, we handle collisions with spheres. The main idea here is if the point mass intersects with or is inside the sphere, then "bump" it up to the surface of the sphere.
Here are some results screenshots:



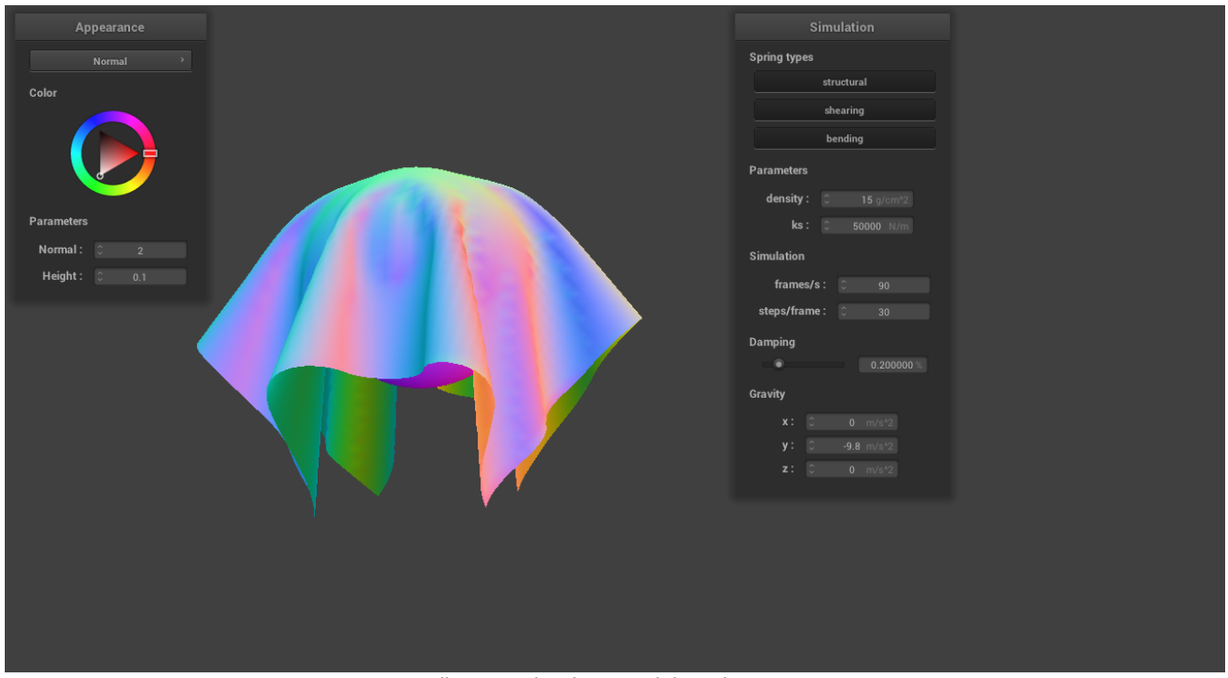
Collisions with spheres with default ks = 5000.



Collisions with spheres with default ks = 5000.

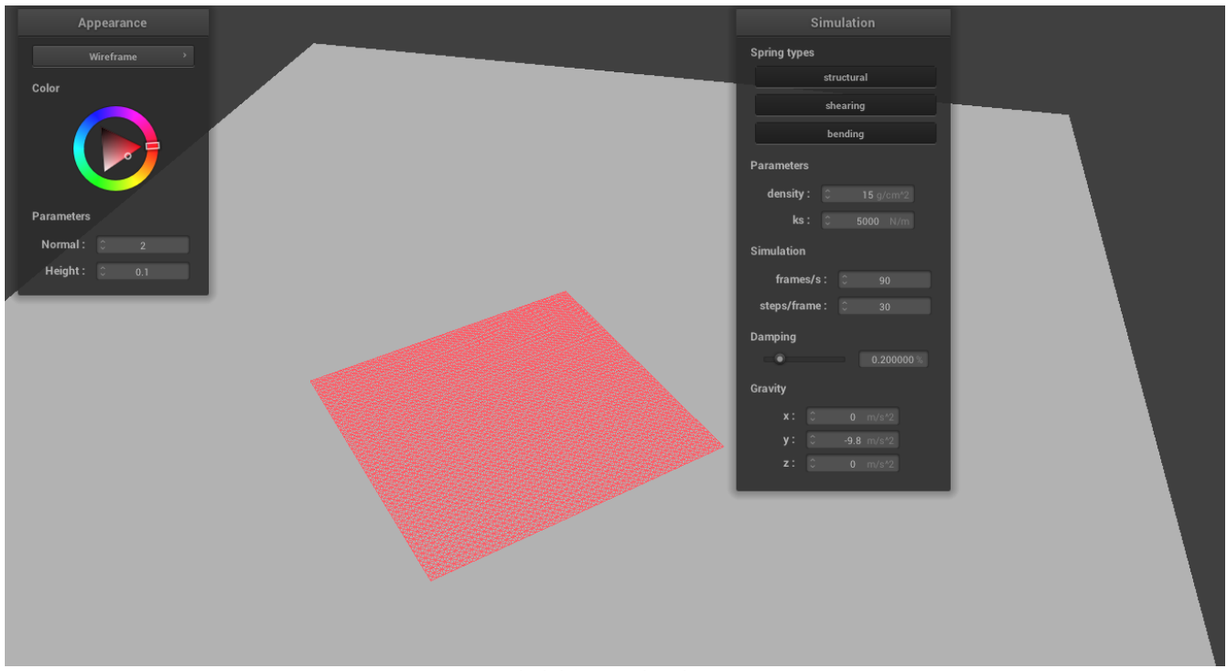


Collisions with spheres with small ks=500.

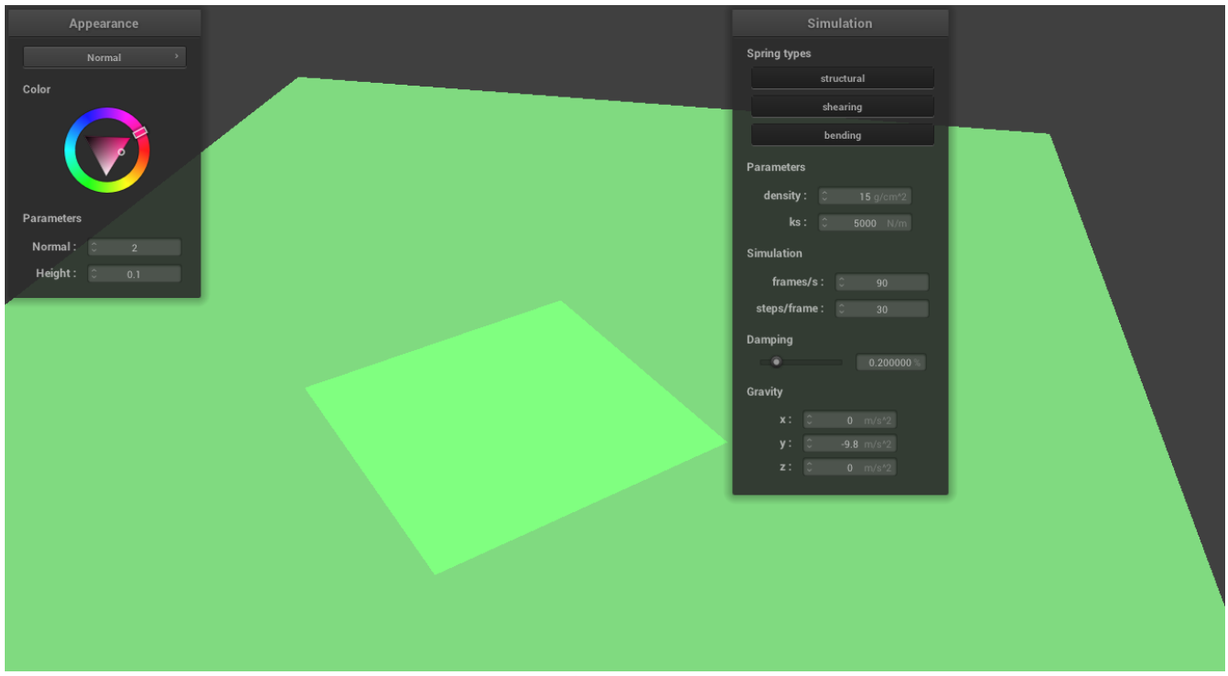


Collisions with spheres with large ks=50000.

We can see that there are differences when ks are different. When ks is larger, the cloth is spreaded out and when ks is smaller, the cloth is closer to the sphere compared to the large ks. This is because if we want to balance the force caused by gravity, when ks is large, the cloth can spread out to keep balance.



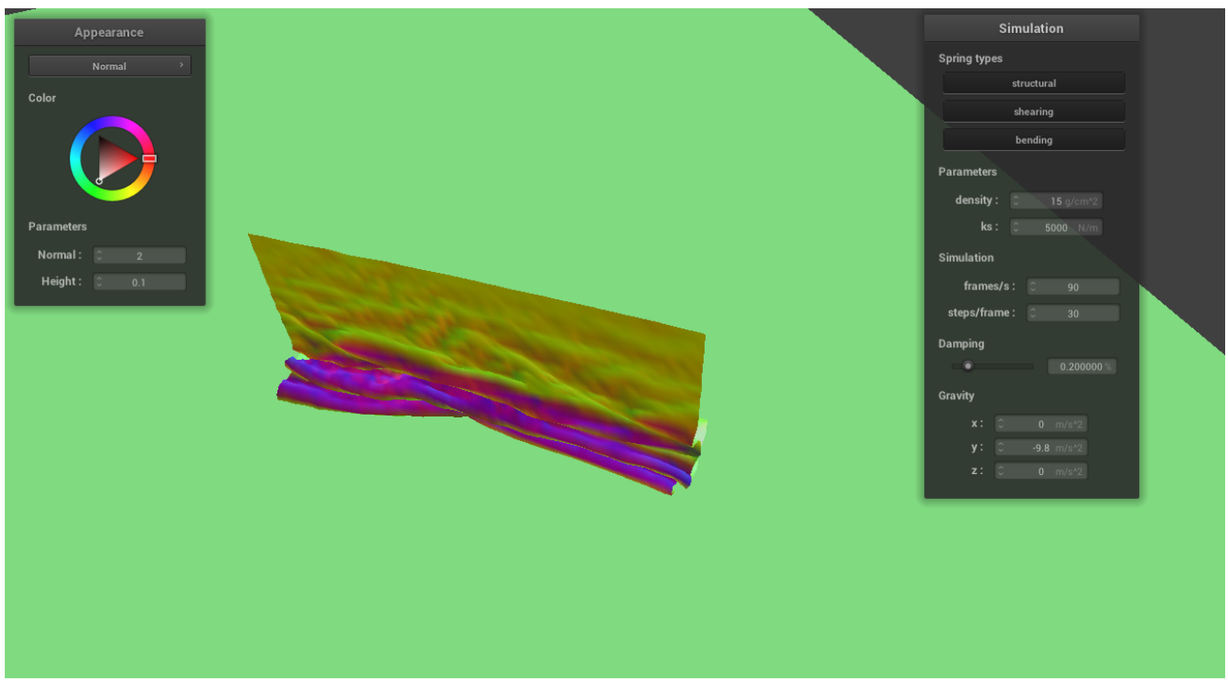
cloth lying on the plane



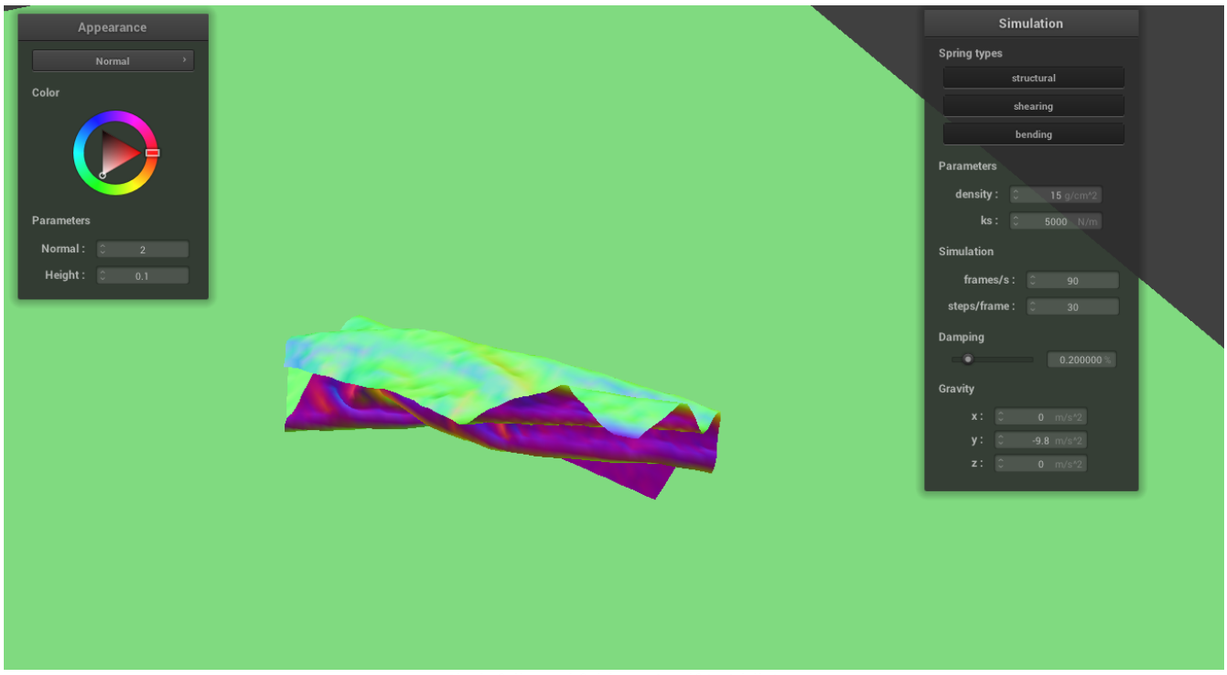
cloth lying on the plane

Part 4: Handling self-collisions

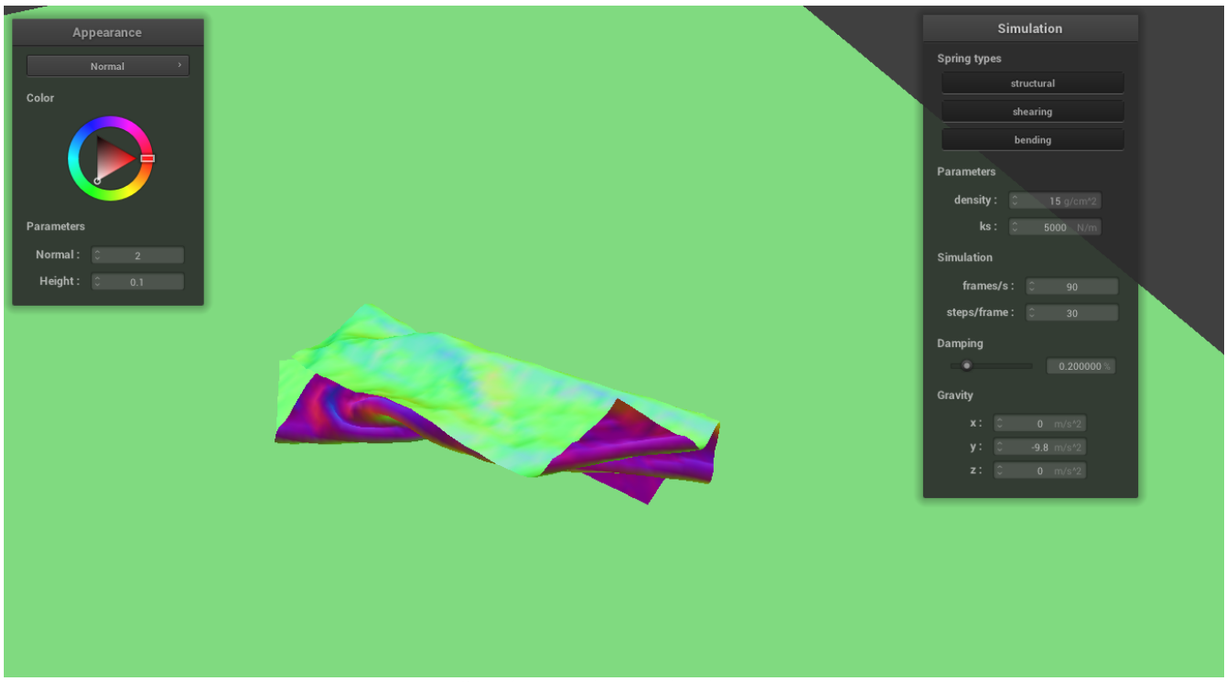
If we have a scene in which the cloth falls on itself or otherwise folds on itself, the cloth will clip through and behave strangely. The naive method of handling this is to loop through all pairs of point masses, compute the distance between them, and then apply a modifying force to the two point masses if they are within some threshold distance apart. However, this will take a long time. In this part, what we do is to implement spatial hashing. At each time step, we build a hash table that maps a float to a mass position. Then we only need linear time to handle this.
Some screenshots that record how the cloth falls and folds on itself are as follows:



Cloth falls and folds on itself (begin).



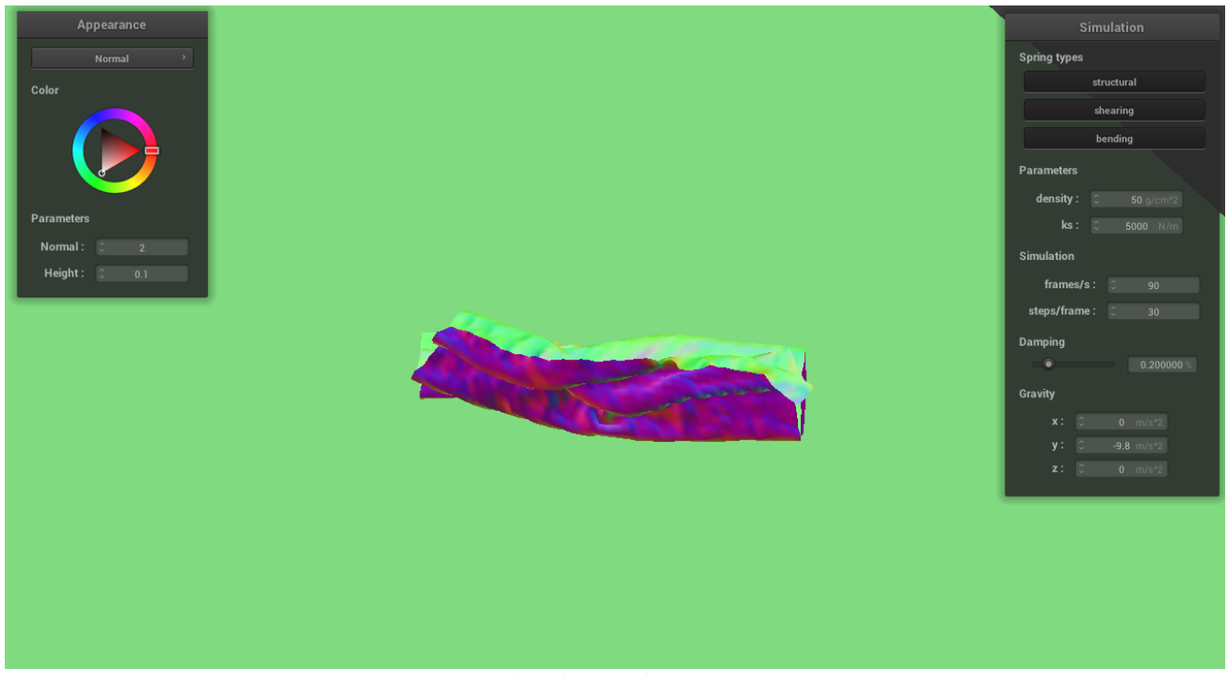
Cloth falls and folds on itself (middle).



Cloth falls and folds on itself (end).

Now let me show how **density** and **ks** affect this process.
First when we use different **density**, the results are as follows:

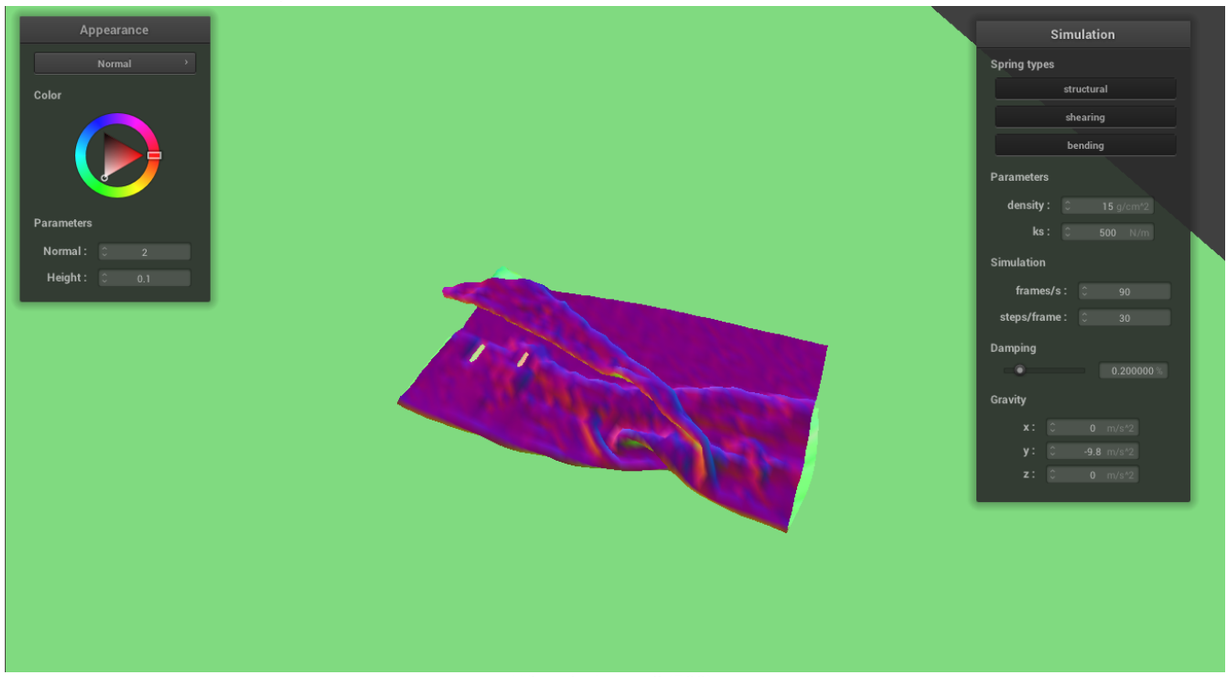
When density is small (5).



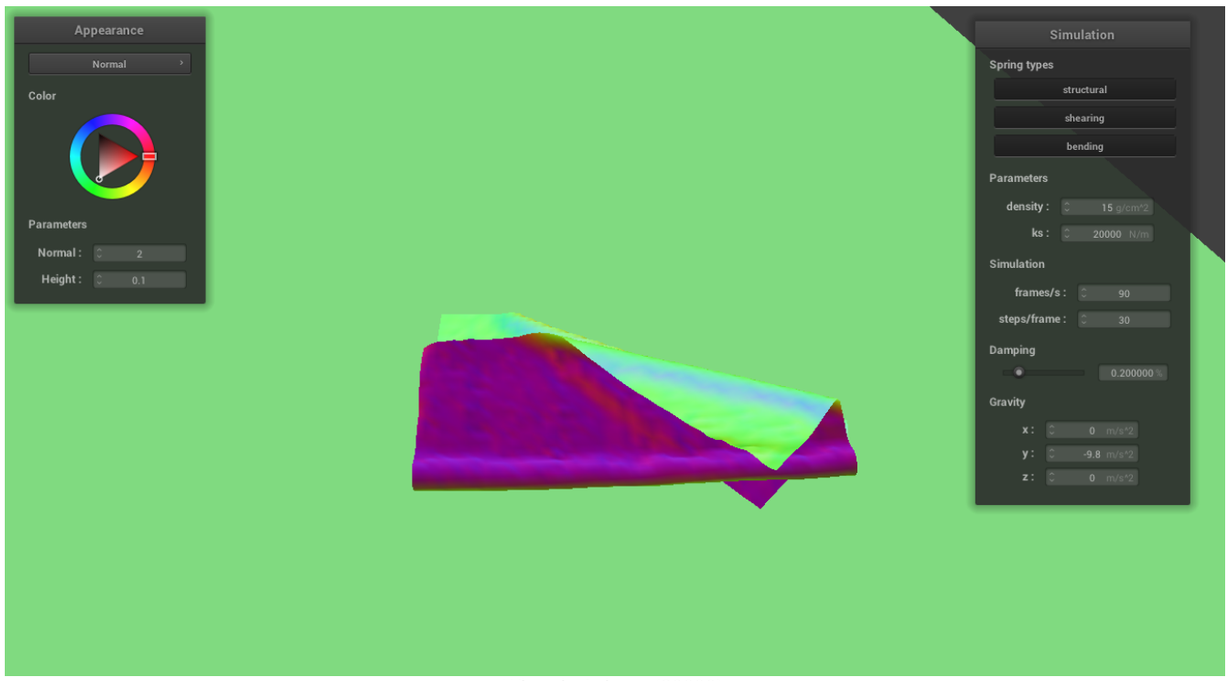
When density is large (50).

As we can see, when density is small, the cloth is more spreaded out, when density is large, the cloth is more likely to fold into itself. This is because when density is small, mass will also decrease, which increases our accelerations back in integration in part 2 and change the spring forces, thus spreading out.

Then when we use different **ks**, the results are as follows:



When ks is small (500).



When ks is large (20000).

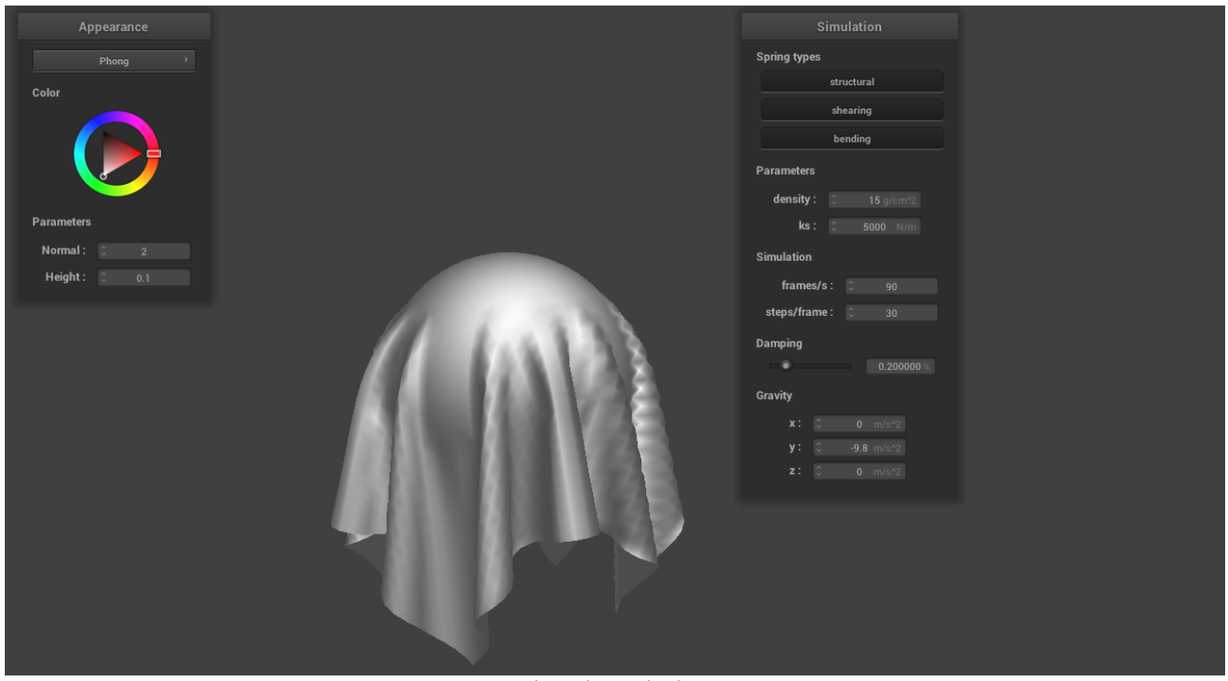
We can see that with small ks, the cloth has more wrinkle. With large ks, the cloth is more flat. This is because that ks controls the stretchiness of the cloth.

Part 5: Shaders

Shader programs are isolated programs that run in parallel on GPU. It takes in vertices in 3D space as input and output the pixel values in the image. In this part, we use GLSL to write code of our shaders to create lighting and material effects.
We will be dealing with two basic OpenGL shader types:**vertex shaders**, **fragment shaders**. Vertex shaders generally apply transforms to vertices, modifying their geometric properties like position and normal vectors, writing the final position of the vertex to gl_Position in addition to writing varyings for use in the fragment shader. Then the fragment shaders take in geometric attributes of the fragment calculated by the vertex shaders, compute and write a color into out_color. This more straightforward in rasterization pipeline. After vertex shaders, we have vertices and triangles positioned in screen space. Then the fragment shaders take the fragment input to calculate the pixel valuesin the image.
Now let me show some of different lighting and material effects.
The first one is **diffuse shading**

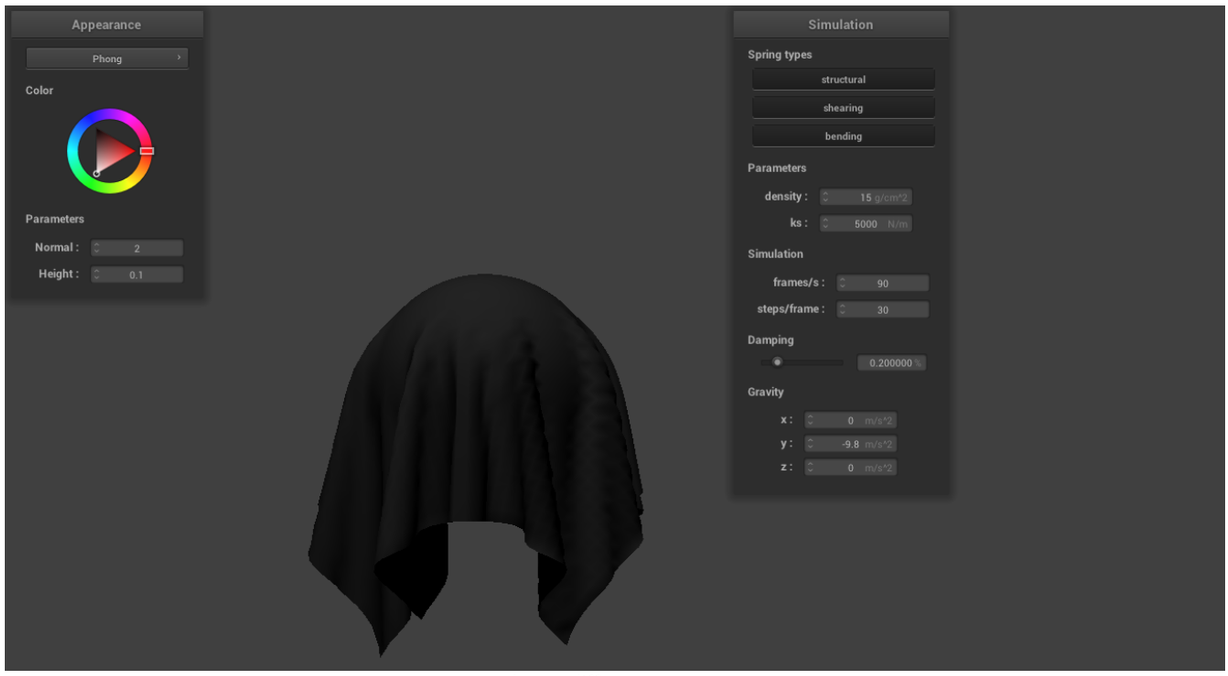
diffuse shading

Then I implement Blinn-Phong shading. Blinn-Phong contains three parts: ambient component, diffuse component and specular component. The diffuse part is the same as last task, and ambient component is a constant vertor which light all parts of the object using the same component and doesn't depend on anything. Specular component contains the information that how close you are to the mirror reflect direction. The final result is as follows.

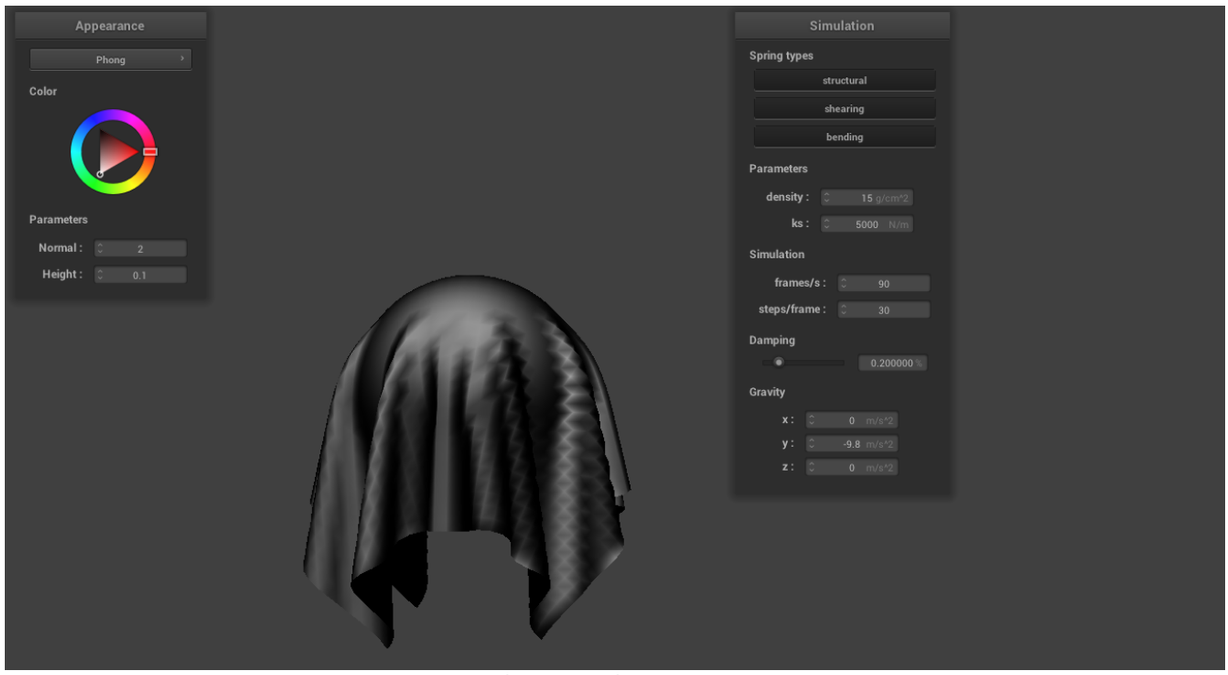


Blinn-Phong shading

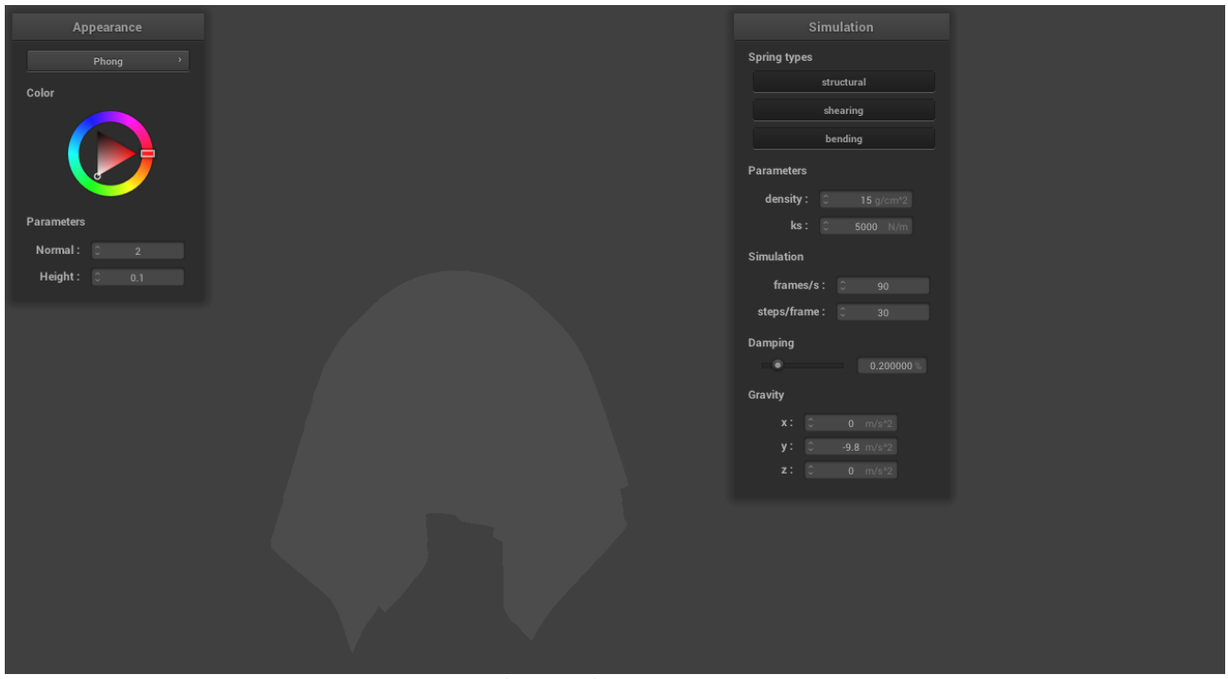
To compare the effect of ambient component, diffuse component and specular component, the following screenshots are using each component.



only use diffuse component

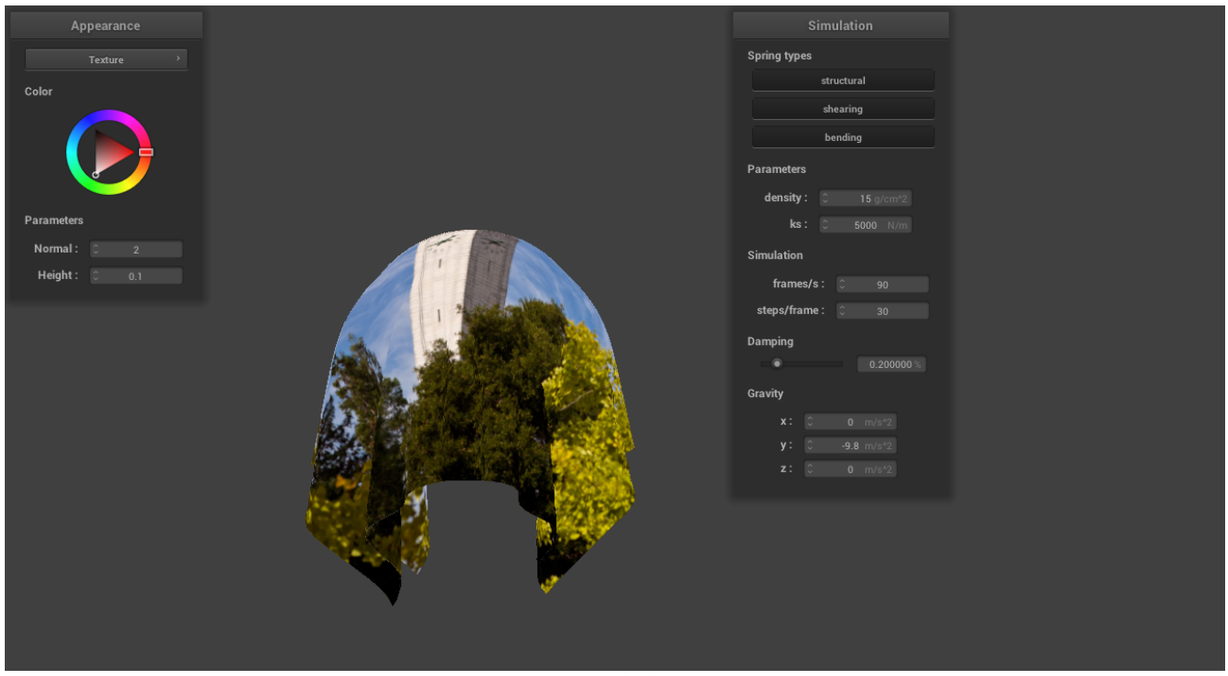


only use specular component

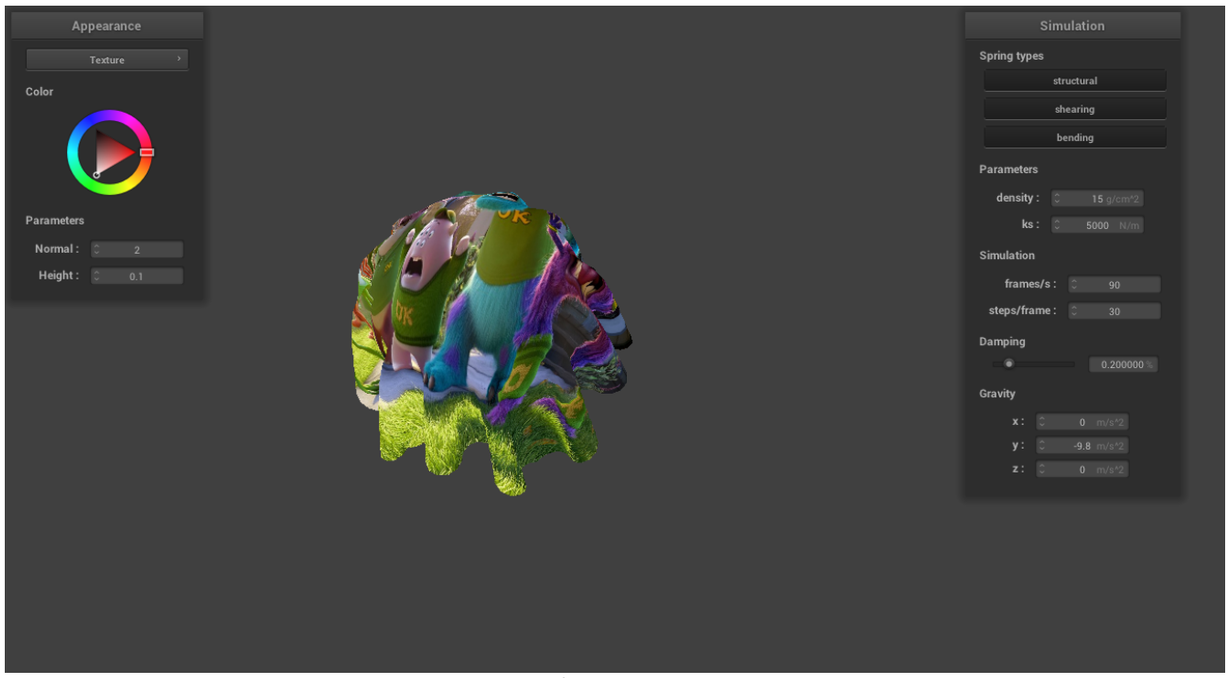


onlu use ambient component

Then let me show two screenshots that use different texture.



amazing!

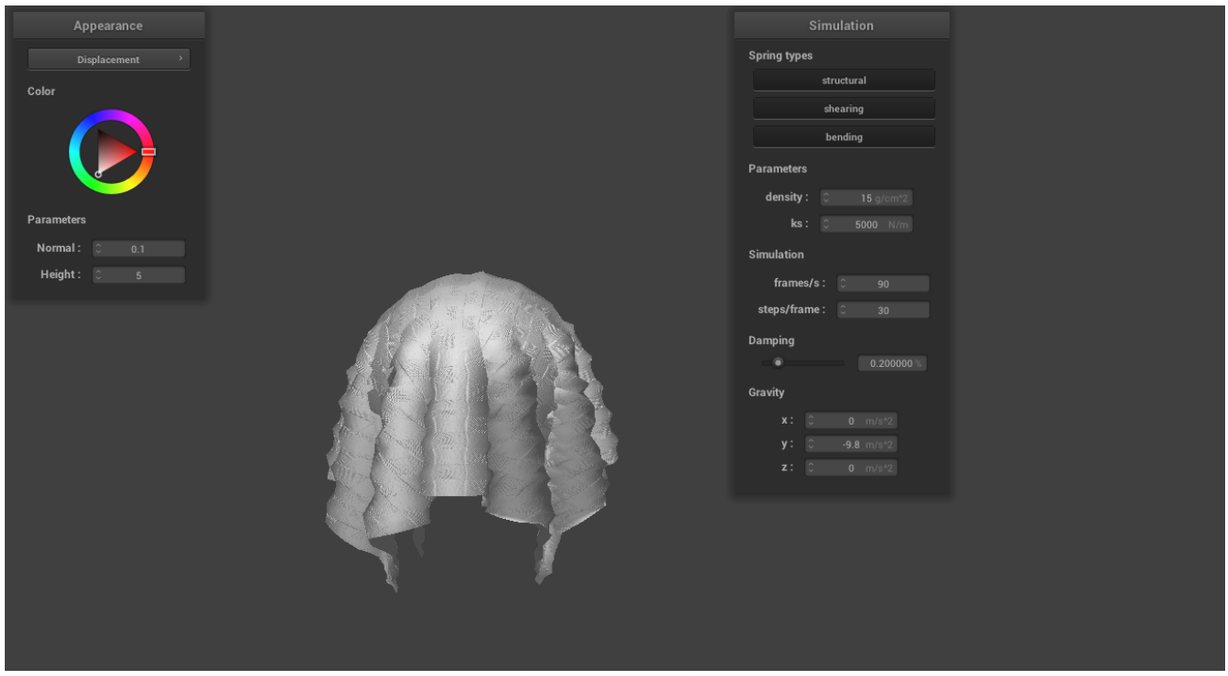


the monsters

Then let me show bump mapping:

bump mapping

Here is displacement mapping:



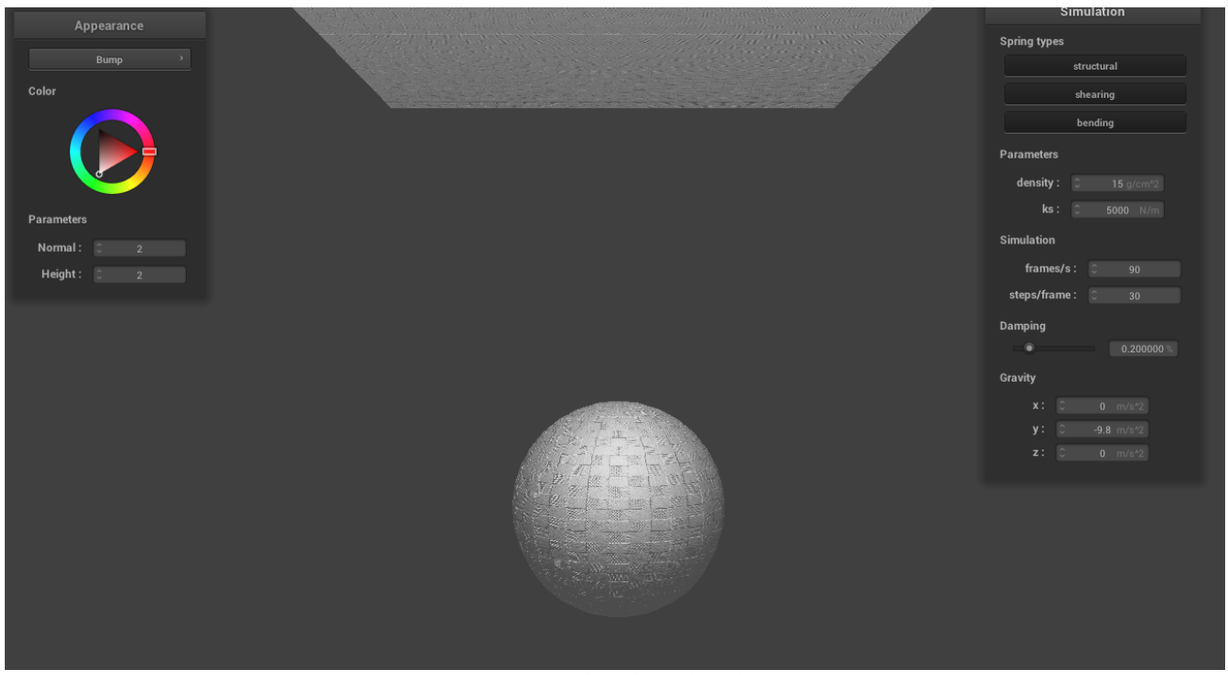
displacement mapping

The main idea of bump mapping is given a vector in object space, we can transform it into back into model space by multiplying by the tangent-bitangent-normal (TBN) matrix. And TBN=[t b n], where t is the tangent vector, n is normal vector, b=cross_product(n,t). And the main idea of displacement mapping is to displace the vertex positions in the direction of the original model space vertex normal scaled by the u_height_scaling variable based on bump mapping.

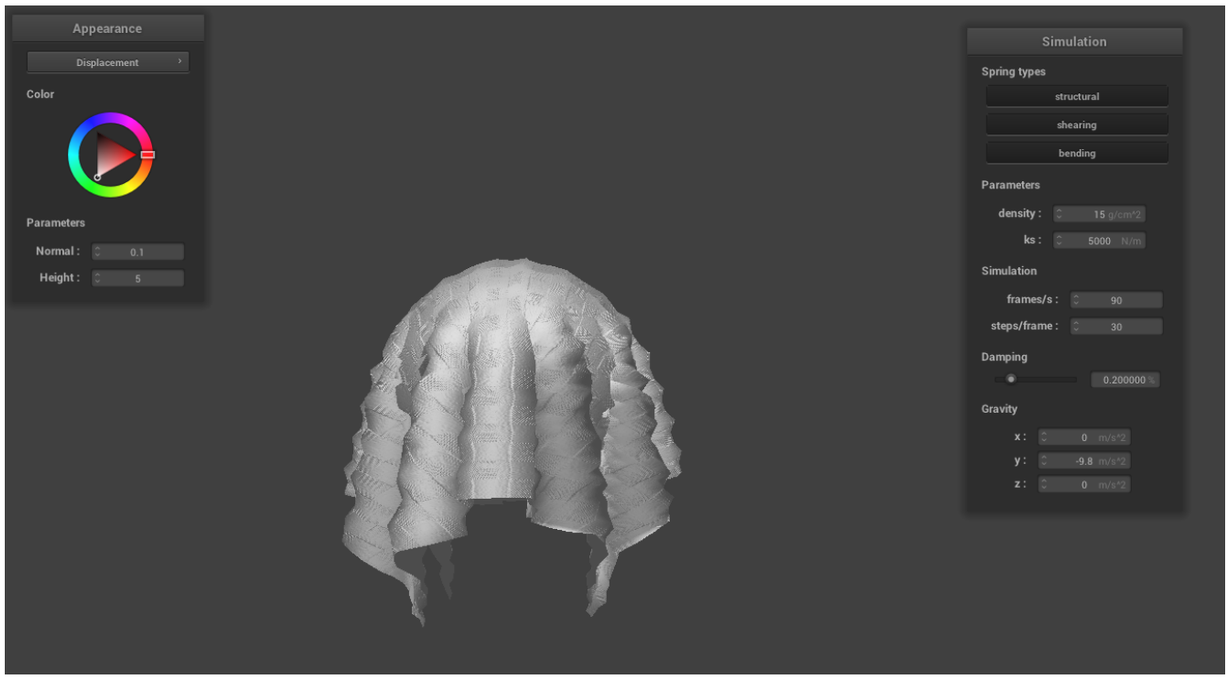
As we can see, bump mapping looks like generating bumps and displacement mapping displaces some of the material.

When using -o 16 -a 16, I get the following results:

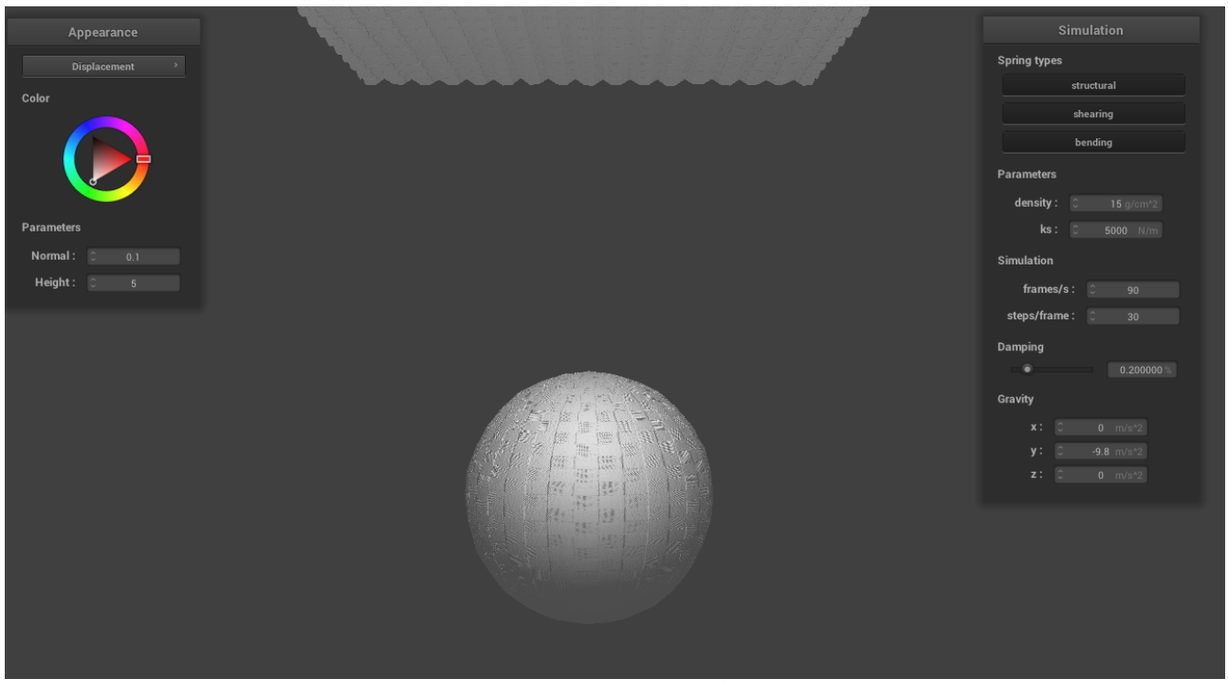
bump mapping with -o 16 -a 16



bump mapping with -o 16 -a 16



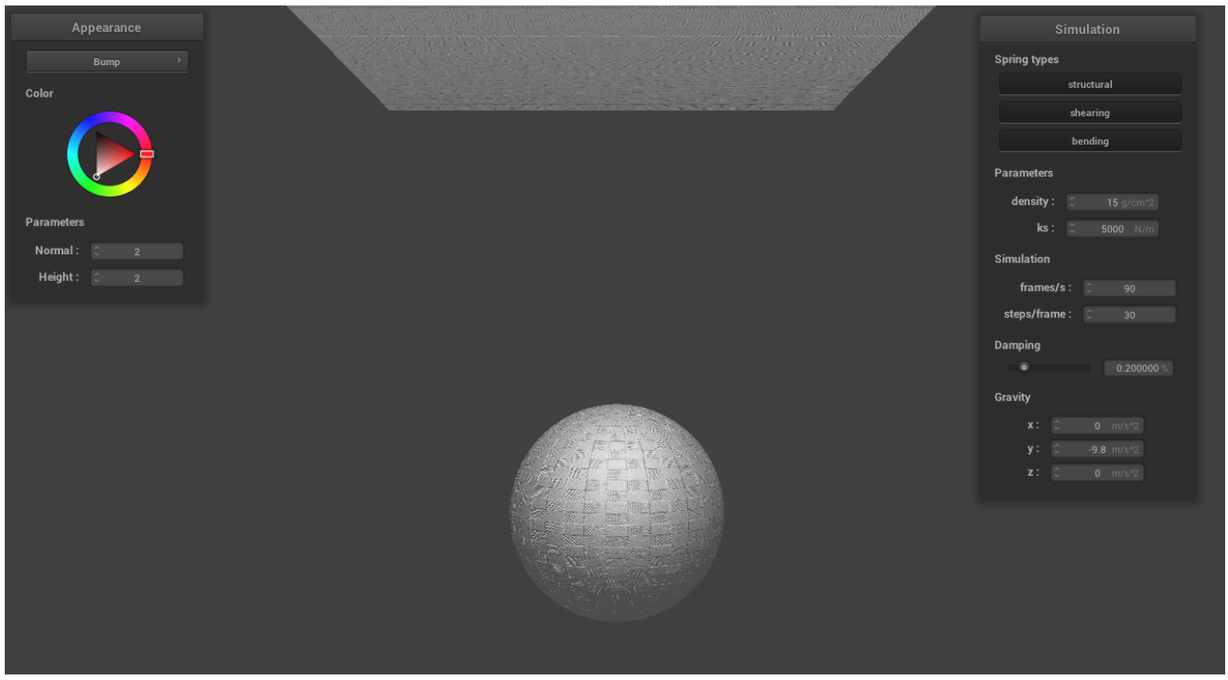
displacement mapping with -o 16 -a 16



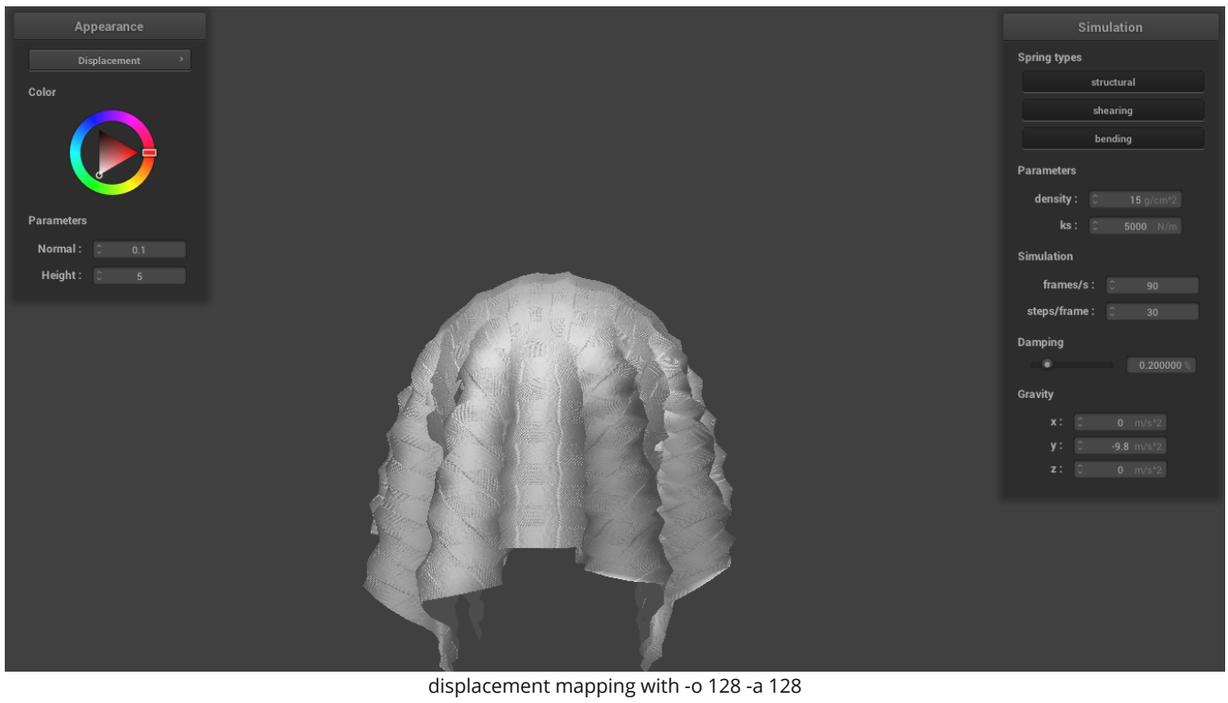
displacement mapping with -o 16 -a 16

When using -o 128 -a 128, I get the following results:

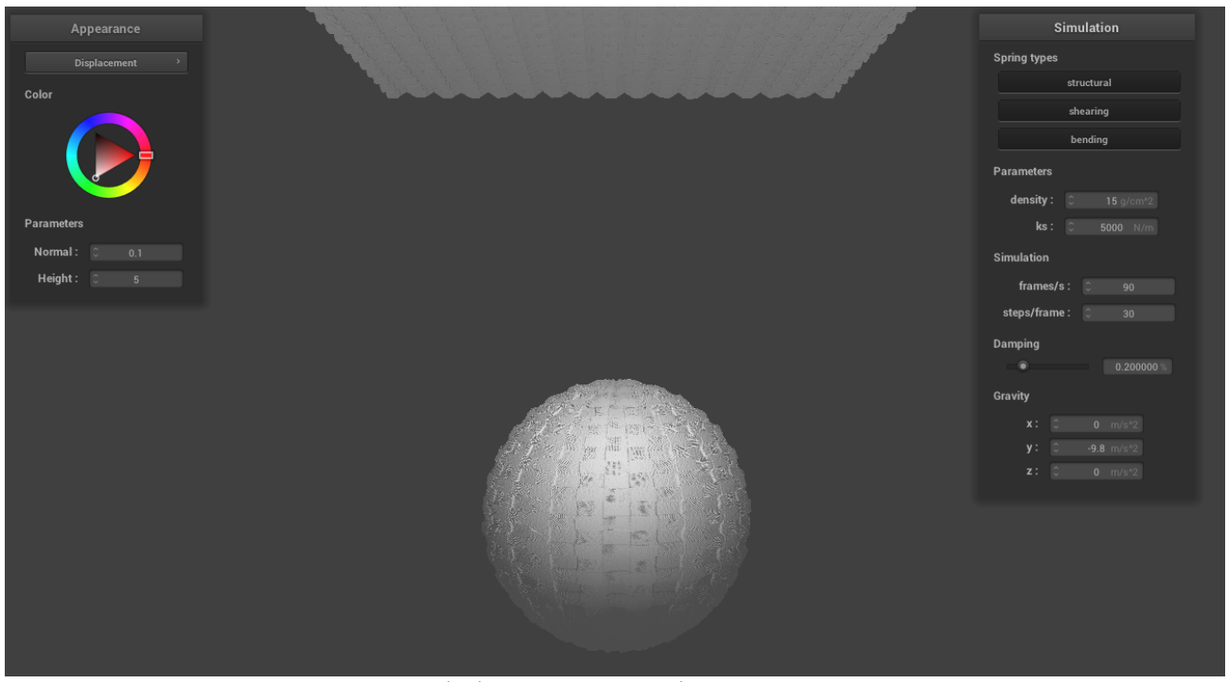
bump mapping with -o 128 -a 128



bump mapping with -o 128 -a 128

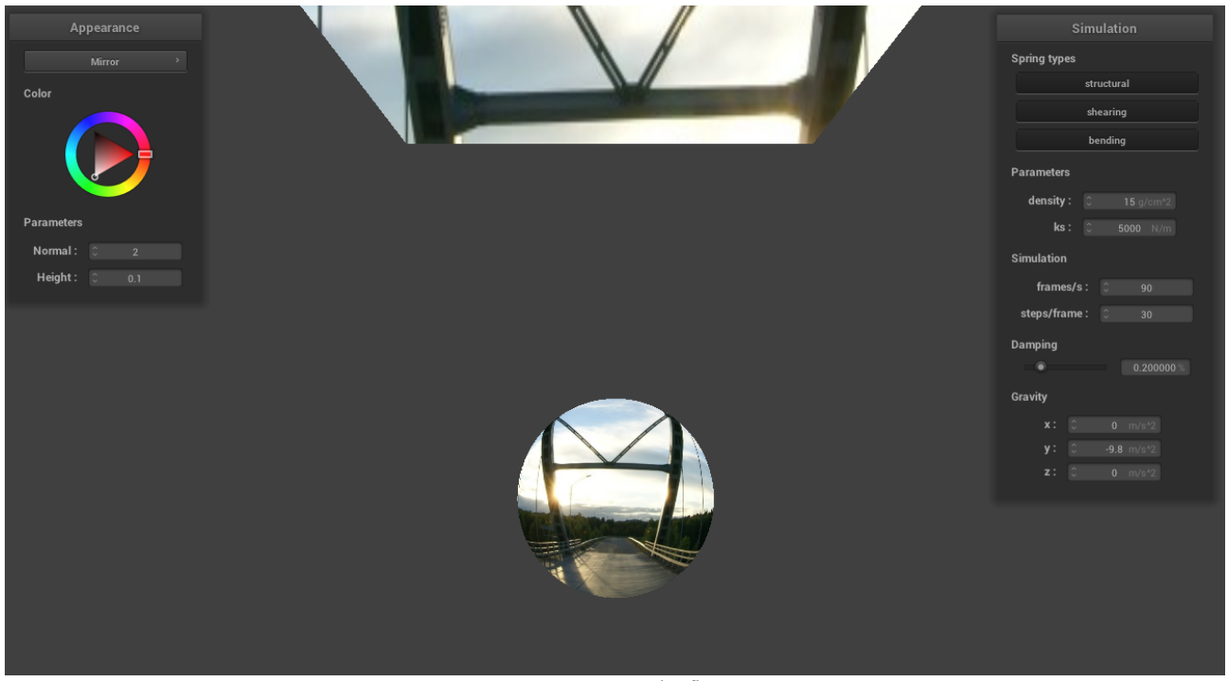


displacement mapping with -o 128 -a 128



displacement mapping with -o 16 -a 16

Then I implement environment-mapped reflections. Here is the result:



environment-mapped reflections