So the writeup got really messed up (doesn't work without CSS/JS) when printed as a pdf, so here is the link: https://ctwobosius.github.io/CS184-P4-Writeup/

PART

April 13, 2021

YEAH WE WERE THOSE GUYS

We made an actual website for a school project, like what, who does that???

Anyways, summary: This was quite a fun project, since we not only saw some cool graphics (looking at you, shaders), but we saw the more interactive side of graphics (animations and simulations, very very juicy). It was cool to see physics and math, two heavy backbones for the raytracing project, in action here, creating relatively realistic cloth movement (with collisions), and then further bringing it to life with shaders.

Also can we get an extra credit point for the website plz lol

A NOTE ON PARTNER COLLABORATION:

Like before, we collaborated by taking turns "piloting," where we discussed approaches, then coded one at a time. However, we didn't always have time to meet together, so sometimes after discussing approaches, one of us would be the ones to actually implement it, without the other person's supervision. We then came back if the other became stuck, trying to debug. For example, one of us noticed the other person had not grouped variables correctly in parenthesis. It went relatively well, and we've been working together for 4 projects so we had almost no trouble dealing with merge conflicts. We learned more about remote software development, debugging each other's code, and writing more understandable code.

A NOTE ON DEBUGGING:

Luckily since there was a lot of interactivity, bugs were a little easier to squash. For example, we had accidentally projected the normal onto a vector rather than the other way around for plane collision testing, which caused the cloth to take a

nosedive off the side, lul. Working with vectors did make it easier to see that our correction vector was bad, since with just checking if we had collided, the cloth stopped right above the plane, which was fine. We did have a few more C++ related bugs, since this project was a little more involved on that front. For example, for auto vs for auto& causing point masses to not update, since it was passing around a copy rather than the actual reference, but overall this wasn't as hard as raytracing to debug (dunno if anything is gonna top that, haha).

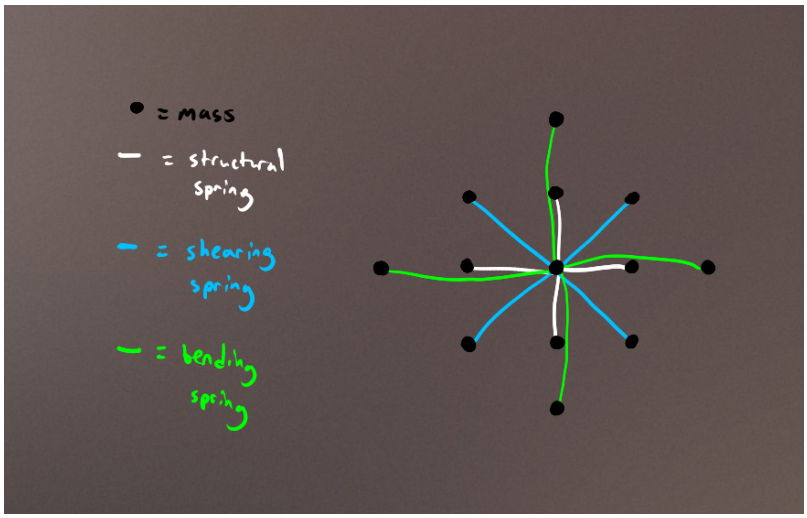
PART

We're springing the massive trap, aha! (Was that a pun on setting up the data???)

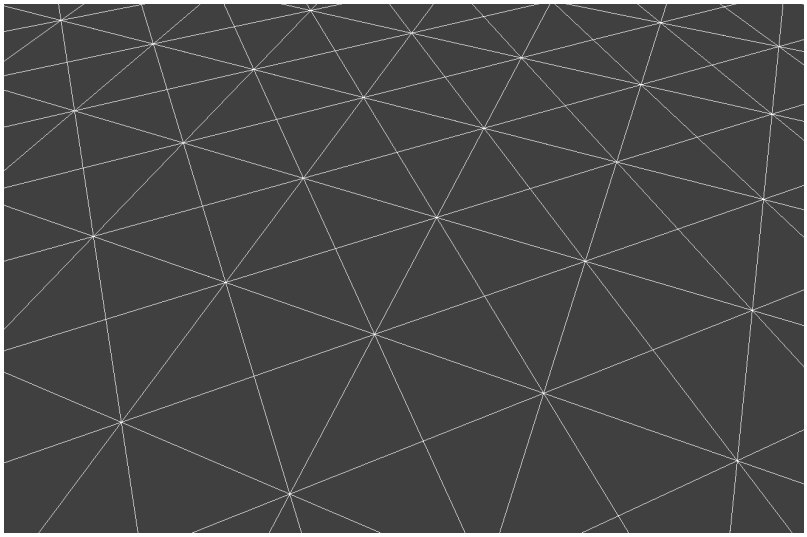
In this part, we set up the basic model for our cloth

that allows us to build upon it. We built a model based on a grid of masses and springs where the cloth is divided up into evenly spaced point masses connected with springs. To implement this, we first created the correct number of masses based on our cloth dimensions, assigned them positions (with a small offset if the cloth is vertical), and added those point masses in row-major order to a **point_masses** vector. Pinned masses were also set up. Then, springs with specific spring types were created between each appropriate pair of point masses (with pointers).

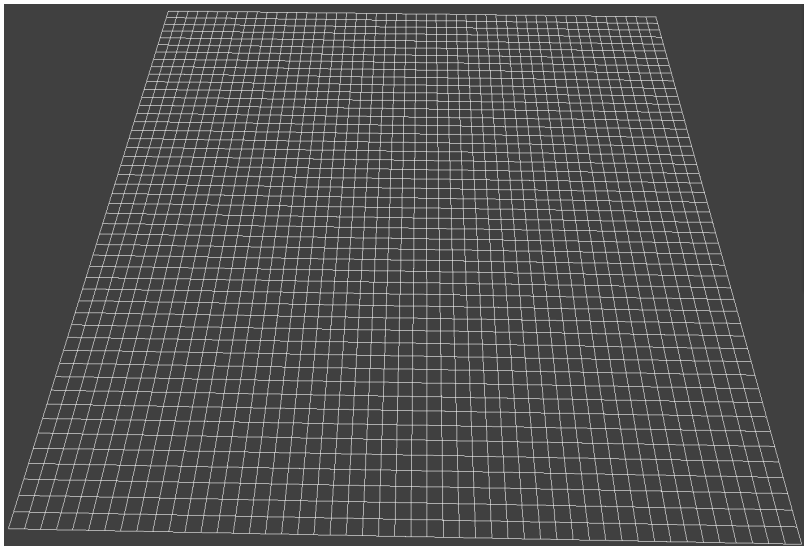
Here's how a mass at the center needs to connect to neighbors (via springs):



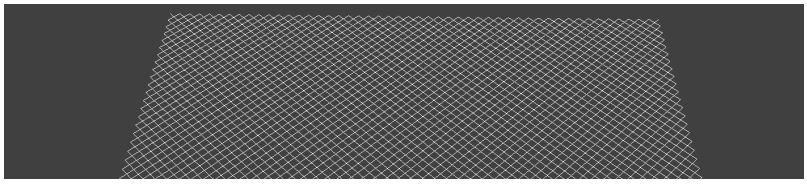
Here is a zoomed image of the grid in action:

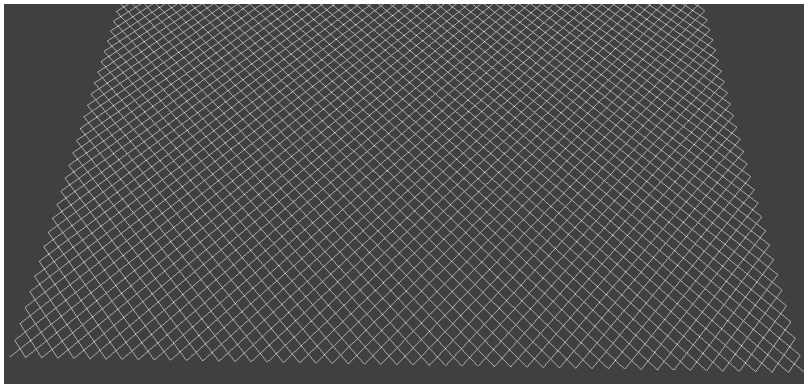


Here is without shearing constraints:

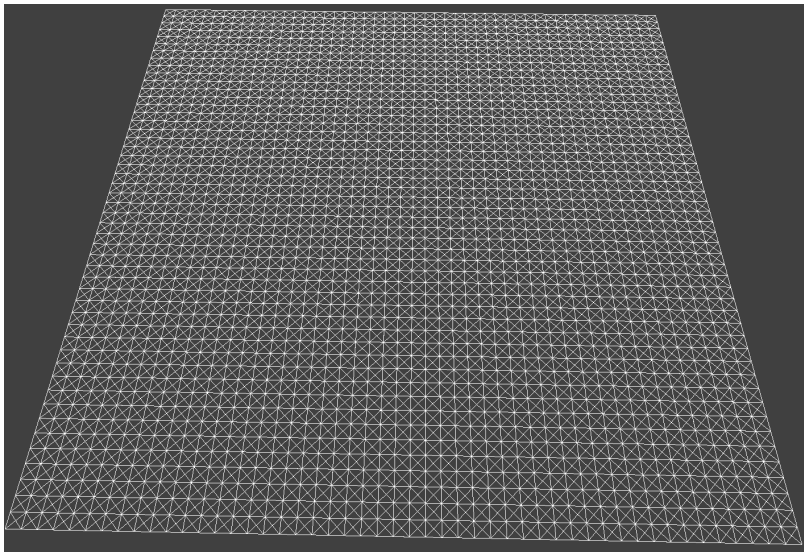


Here is with only shearing constraints:





Here is with everything enabled:

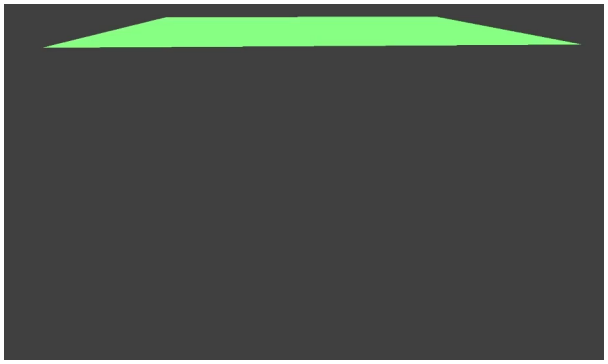


PART

(VERLET) INTEGRATION

Make that cloth dance in the spring! (haha very pun,

much fun)



For this part, we simulate cloth movement. First, we sum all forces acting on each point mass. This is achieved using physics, such as Newton's famous F=ma, along with Hooke's law (for springs):

$$F_s = k_s * (||p_a - p_b|| - l)$$

where ks is a spring constant, where larger values simulate "stiffer" springs (return to rest faster), p_a and p_b are the positions of masses attached to the springs, and l is the spring's rest length.

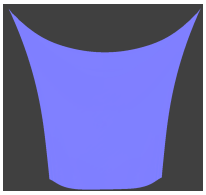
Then, we use Verlet integration to compute a point mass's position at each timestep of the simulation according to the forces applied to each mass. Mathematically, this is

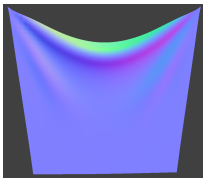
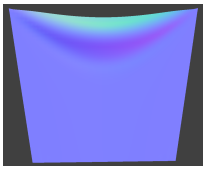
$$x_{t+dt} = x_t + (1 - d) * (x_t - x_{t-dt}) + a_t * dt^2$$

where t+dt is the next time step, d is a damping constant (more means more damping/less movement), t-dt is the previous time step, a_t is acceleration (calculated using F=ma), and dt is the time step.

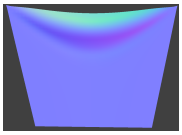
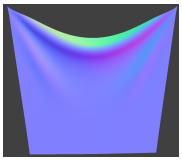
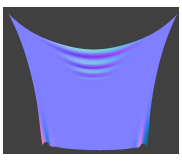
To prevent springs from becoming unreasonably deformed, we also add constraints to each spring so that its length is at most 10% greater than its rest_length at the end of any time step. We make sure that the spring's direction is preserved, for our implementation. After implementing this part, we can now simulate our sheet of cloth.

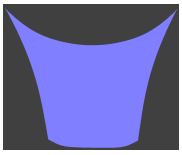
Here are various ks values (N/m):

KS	IMAGE	DIFFERENCE
1		Cloth is less "stiff" and as a result there are barely any wrinkles, since it just falls straight down without much constraints.

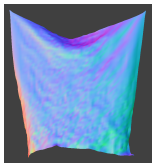
KS	IMAGE	DIFFERENCE
5,000		Default, looks pretty natural, with some folding and wrinkles.
100,000		Cloth is very "stiff" and thus it doesn't go as far from its starting position.

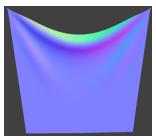
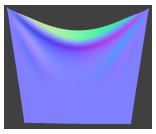
Here are various density values (g/cm^2):

KS	IMAGE	DIFFERENCE
1		Cloth is not "weighed down" as much, so there's barely a curve at the top, with it just slightly deforming.
15		Default, looks pretty natural, with some folding and wrinkles.
1,000		Cloth is "weighed down" much more, so the curve at the top is

KS	IMAGE	DIFFERENCE
		much more pronounced, with the wrinkles being smaller since the weight pulls it down and "stretches" it.
100,000		Cloth is "weighed down" so much that it looks flat, all the wrinkles are practically gone because of the sheer weight of the cloth, lol.

Here are various damping values (%):

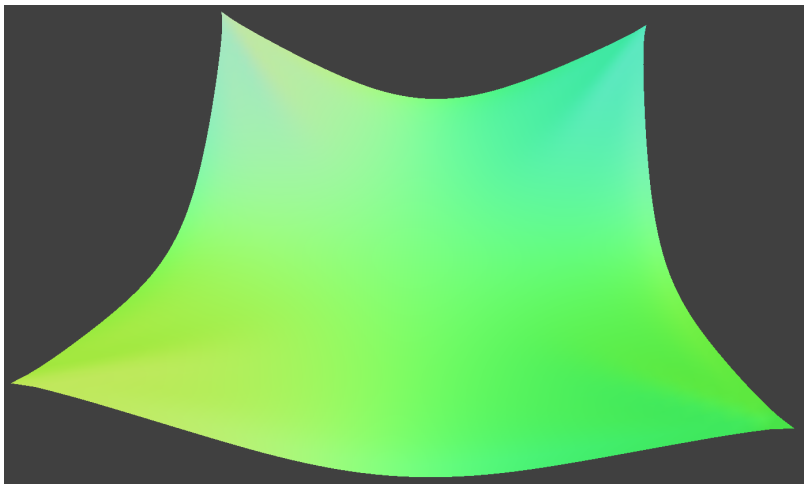
KS	IMAGE	DIFFERENCE
1.15		Cloth's movements aren't as restricted, so it flutters. A lot. Like a butterfly, the slightest movement causes it to go fly away, back and

KS	IMAGE	DIFFERENCE
		forth, since the "frictional effect" of larger damping constants are more or less absent.
20		Default, looks pretty natural, with some folding and wrinkles.
100		Cloth is pretty damped, so it falls very very slowly, and in general, is hard to move. However, it looks the same as default after it settles (which should also be the case once the 1.15% one decides to settle down like a good boi), since this only really affects how it moves, rather than its "convergence" state.

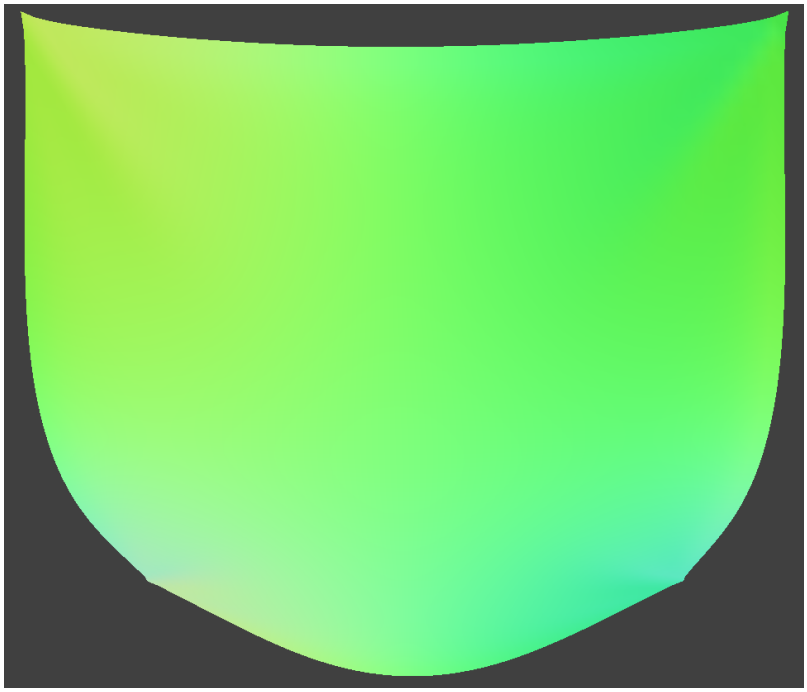
Pinned_4 was rendered using the model's normals and the default parameters (all spring types

enabled, density: 15 g/cm^2, ks: 5000 N/m, damping: 0.2%, gravity vector: (0, -9.8, 0) m/s^2)

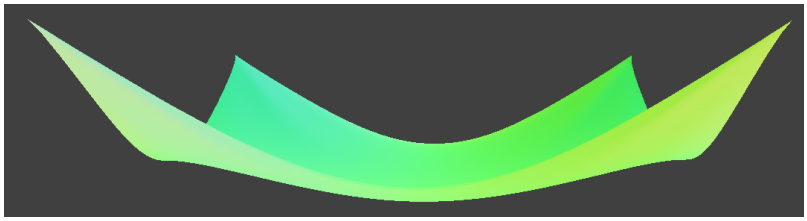
Here is pinned_4's top view:



Here is pinned_4's bottom view:



Here is pinned_4's side view:



PART

OBJECTS

So we made it look like water? .O.

BEHIND THE SCENES

We implemented collisions with spheres and planes. In general, to check for collisions we simply looped over all collision objects and checked if any of our point masses collided with it.

This was a naive approach, and perhaps we could have made it better, using spatial hashing, for example.

For sphere collisions, we used the fact that if a point's distance to the sphere's center was less than its radius, it collided with the sphere (we further optimized this since we could use the radius and distance squared, which avoided two square root operations per check).

We then bumped the mass's position to be slightly above the sphere's surface, conserving the direction to the sphere's center.

For plane collisions, let P be a point on the plane. We checked if the dot product of *the vector from P to the mass's position* with *the plane's normal* was not the same sign as the dot product of *the vector between the mass's previous position and P* with *the plane's normal*, and if so, the mass had gone through the plane, since the normal was roughly the same direction for one but the opposite direction for the other.

We then used the projection a onto b:

$$\frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{b}\|^2}\mathbf{b}$$

where a was the vector from P to the mass's last position, and b was the normal (this worked since both vectors are relative to the point on the plane).

This gave us the vector that, when subtracted from the mass's position, was the collision point. We then bumped this projection to be slightly before the collision point, scaled by friction, then used

that to move the mass to just right before the mass actually touched the surface.

PRETTY PICTURES

Various ks values (N/m):

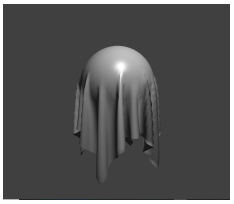
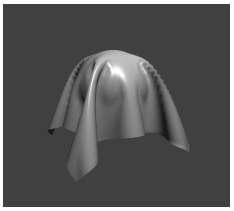
IMAGE	COMPARISON
	**500:** Compared to the default cloth of 5000 N/m, the cloth closely hugs the sphere so that the top half of the cloth looks like just like the sphere itself. It also has a lot more folds and wrinkles and resembles a wet paper towel.
	**5,000:** Default setting. Cloth looks stiffer than the 500 N/m cloth.
	**50,000:** Cloth is very stiff and has less folds and wrinkles than the 5000 N/m cloth. The folds of the cloth also "reach

IMAGE	COMPARISON
	out" more, pointed away from the sphere, since the stiffness prevents it from bending as much.

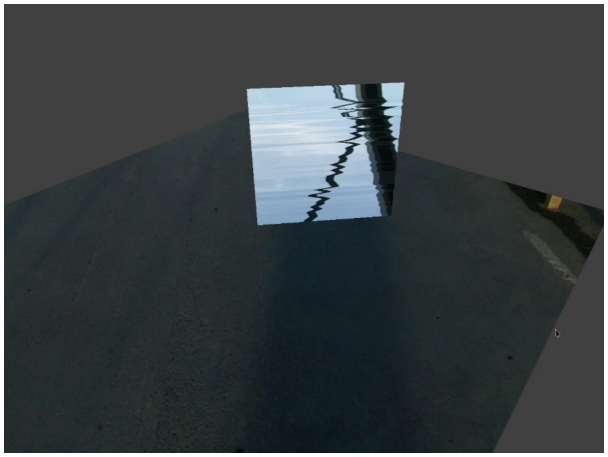
Cloth with Campanile texture lying on top of plane:

Cloth with normals shading lying on top of plane:

PART

ld

it be self-introspecting?



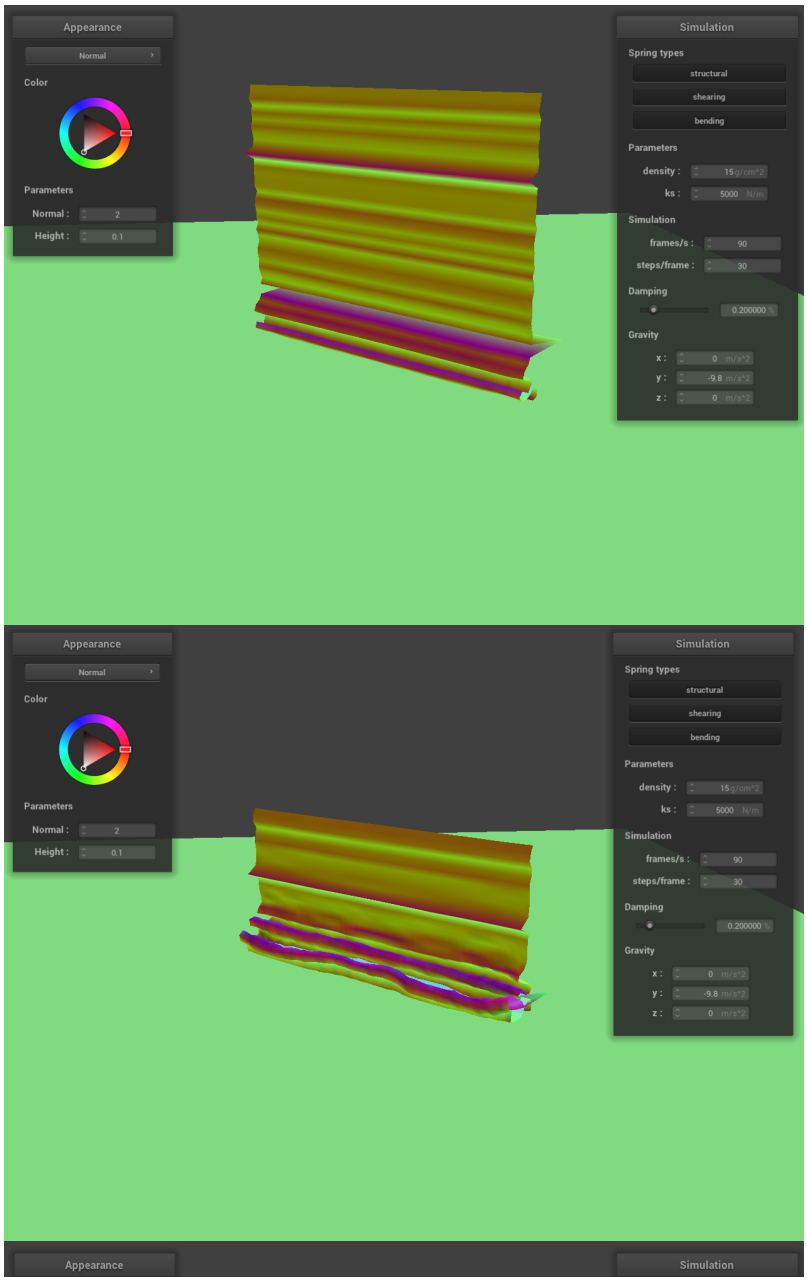
In this part, we implemented self-collision for the cloth so that when it falls or folds in on itself, it won't clip through. For this part, it was too slow to naively check each point against every other point; thus, we used spatial hashmaps to speed up these checks.

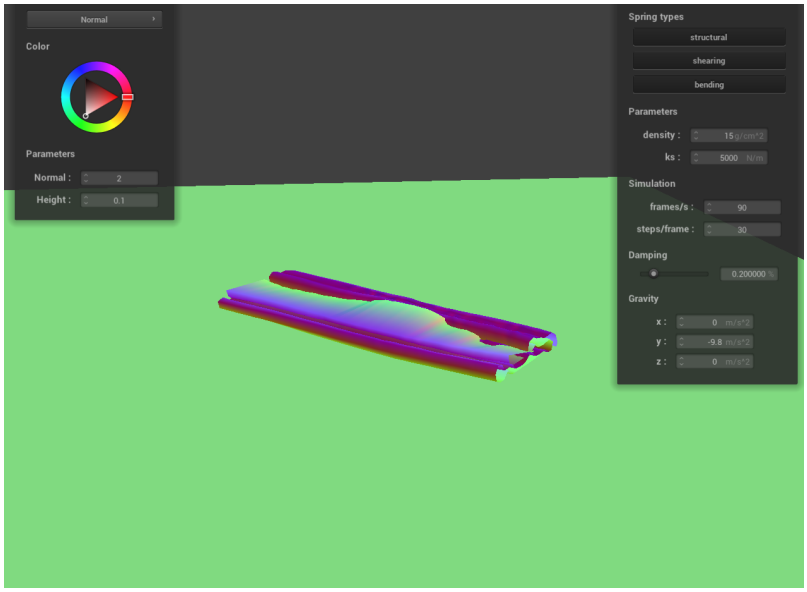
Our scheme consisted of dividing our space into 3D boxes, grouping all masses within a box into a single hash group, then to actually hash the mass, we floored the mass's coordinates to the nearest box, then hashed that floored position using this post (we tried others but this one was relatively fast).

To actually correct masses, if the mass was less than 2*thickness away from another mass, we

added the correction vector to make the mass 2*thickness away to a cumulative vector sum, then averaged these at the end (and divided by simulation_steps to smooth it out more).

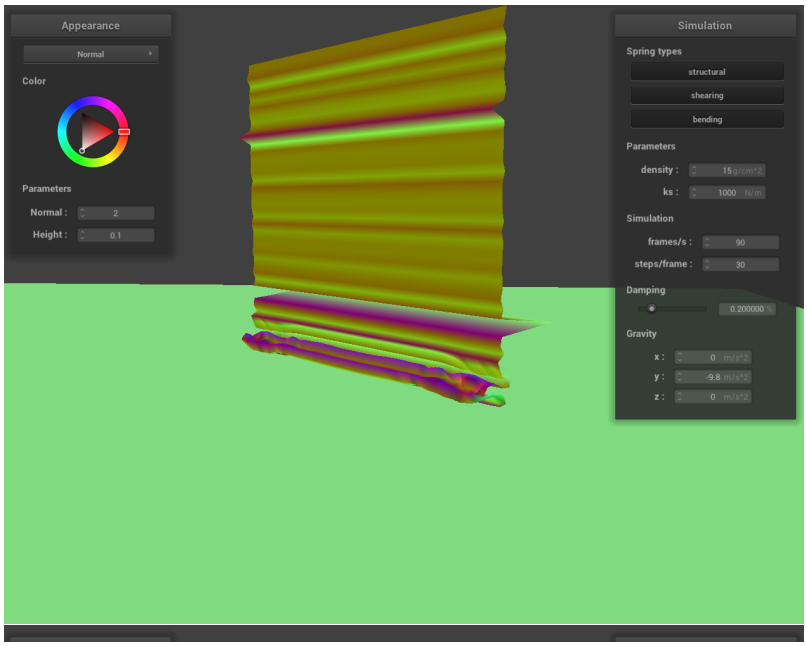
Below we see the correct behavior of the cloth falling in on itself.

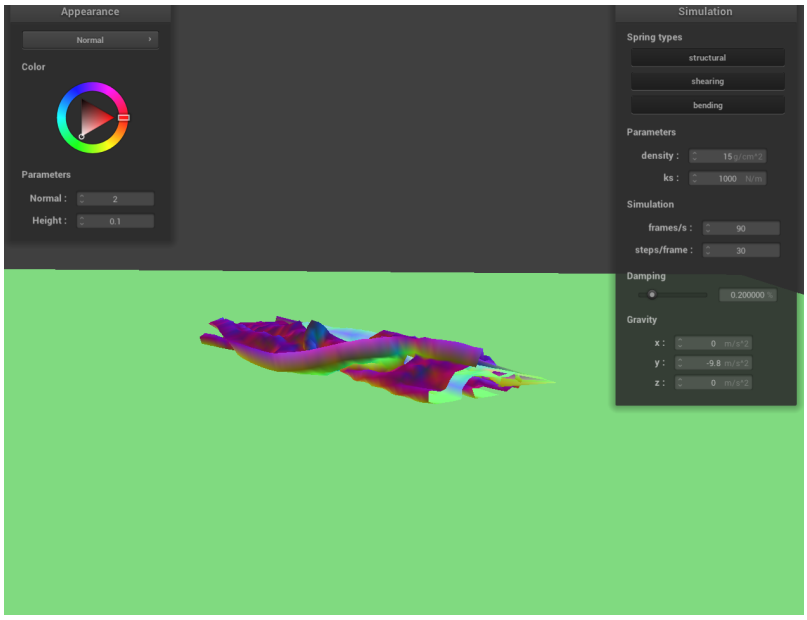




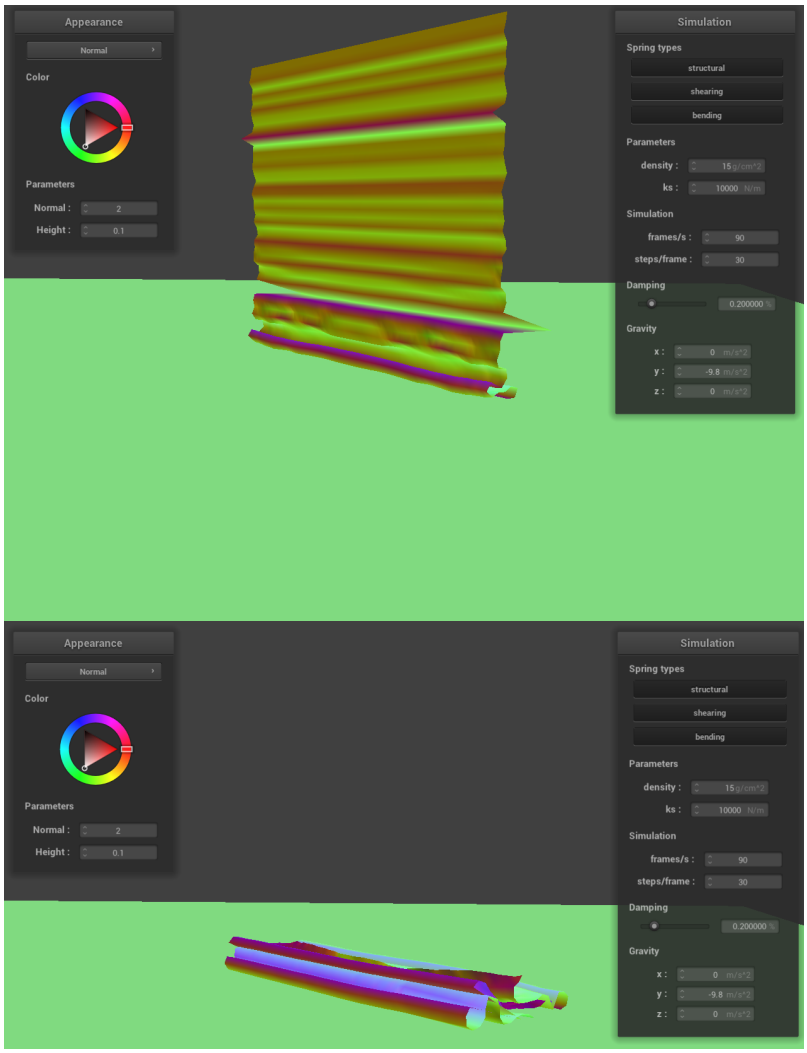
We can also vary the density and the constant ks to affect the behavior of the cloth as it falls on itself. Let's see how changing the ks value affects the cloth below:

Self-collision with the **ks value set to 1,000**:



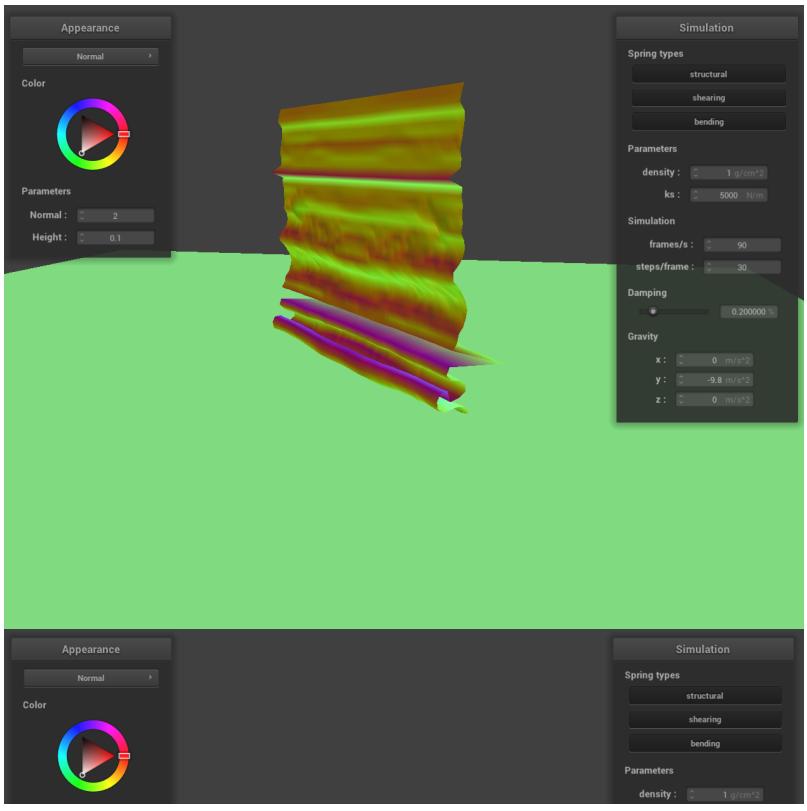


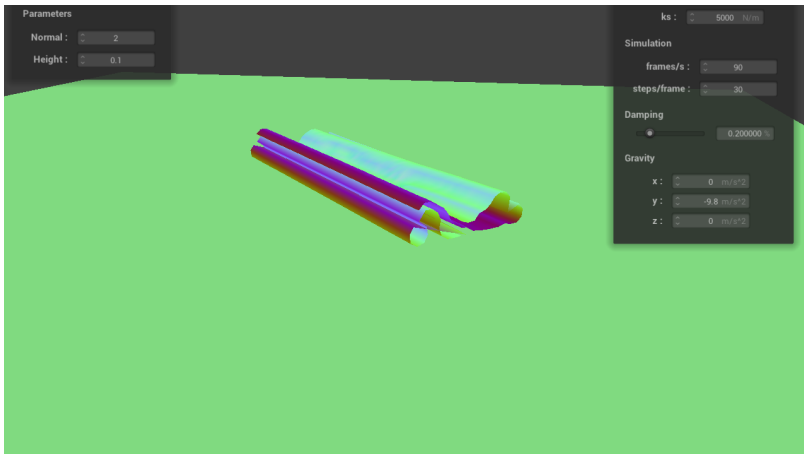
Self-collision with the **ks value set to 10,000**:



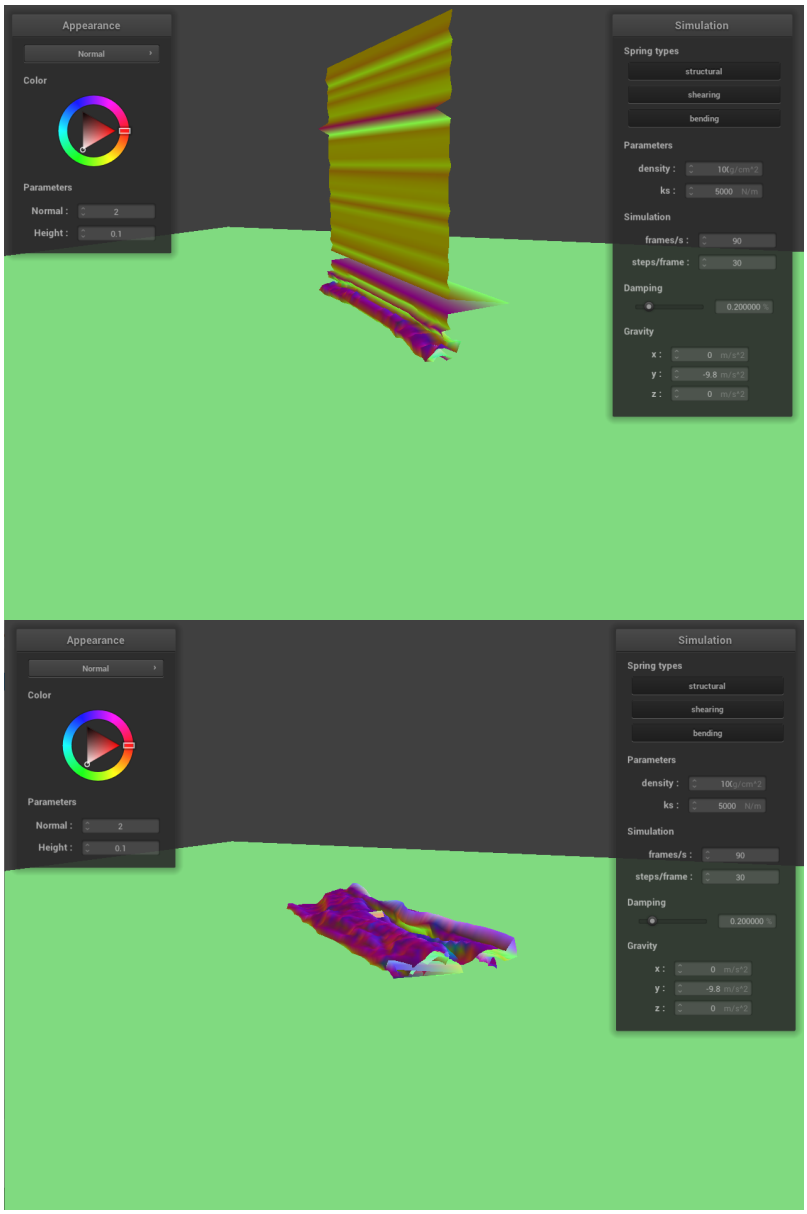
A lower ks value leads to springier cloth. We can see that in the differing collisions above where the cloth with the ks set to 1,000 ends up more spread out and disheveled. The cloth with the 10,000 ks resembles a big sheet of paper and ends up folding up nicely with large curls. Now let's see how changing the density of the cloth affects it:

Self-collision with **density set 1**:





Self-collision with **density set 100**:



The denser the cloth, the heavier it is, which affects how it falls into itself. We see that when the cloth has a density of 1, it ends up folding nicely on top of itself. It seems to resemble aluminum foil as it lies relatively straight and folds into curls. When the cloth has a density of 100, however, it doesn't fold up as nicely and lies on the ground, without any curly bends, resembling a folded up blanket.

PART

In this part, we implemented a few GLSL shader programs. Shaders are realistic lighting programs that run parallel in the GPU instead of in the CPU. It executes sections of the graphics pipeline by taking in inputs and outputting 4 dimensional vectors. As a result, shaders accelerate rendering of

lighting and material effects that would take a long time on the CPU.

GLSL shader programs are composed of two things: vertex shaders and fragment shaders. Vertex shaders apply transforms to vertices and writes to variables used in the fragment shader. Fragment shaders process fragments that are created after rasterization and write out a color for the fragment.

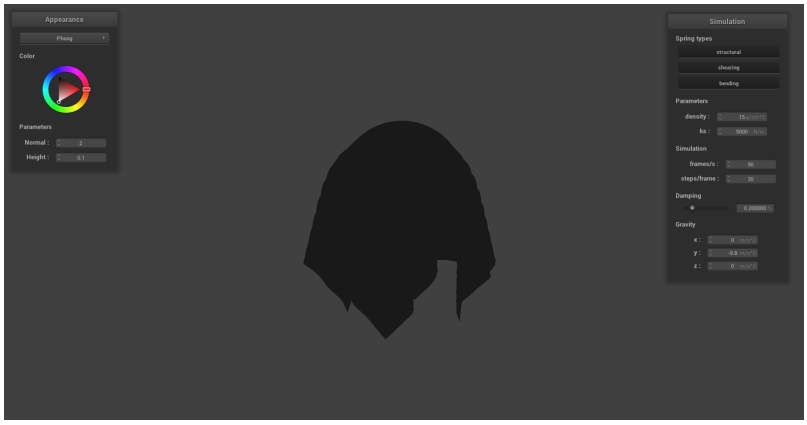
In our first task, we implemented diffused shading by calculating the **out_color** using the diffused shading equation shown in the spec and lecture (using a **diffuse coefficient of 1.0**). Then, we implemented Blinn-Phong shading, which incorporates diffused shading from our first part. The Blinn-Phong model is a lighting model that not only accounts for diffused shading but also ambient lighting and specular reflection to give us a better model of lighting on an object. The use of all three components leads to the following equation for Blinn-Phong:

$$\mathbf{L} = \mathbf{k}_a \, \mathbf{I}_a \; + \mathbf{k}_d \, (\mathbf{I}/r^2) \, \max(0, \mathbf{n} \cdot \mathbf{l}) \; + \mathbf{k}_s \, (\mathbf{I}/r^2) \, \max(0, \mathbf{n} \cdot \mathbf{h})^p$$

For our Blinn-Phong calculations, we used the follwing constant values:

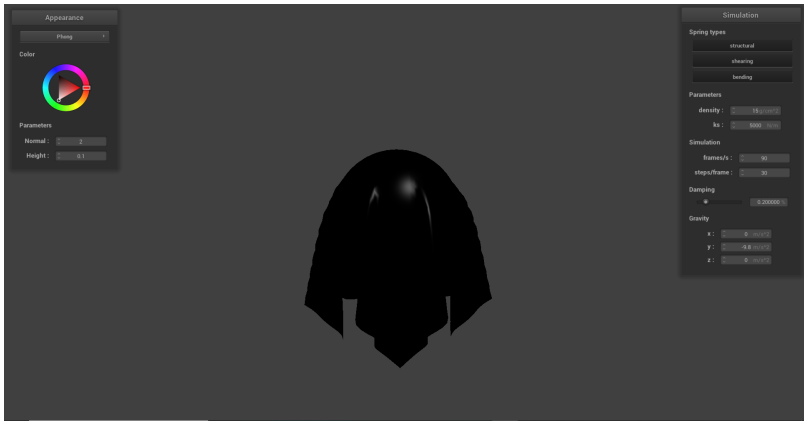
```
ka = 0.1, kd = 1.0, ks = 0.6, p = 80
```

Ambient component of Blinn-Phong:



Diffuse component of Blinn-Phong:

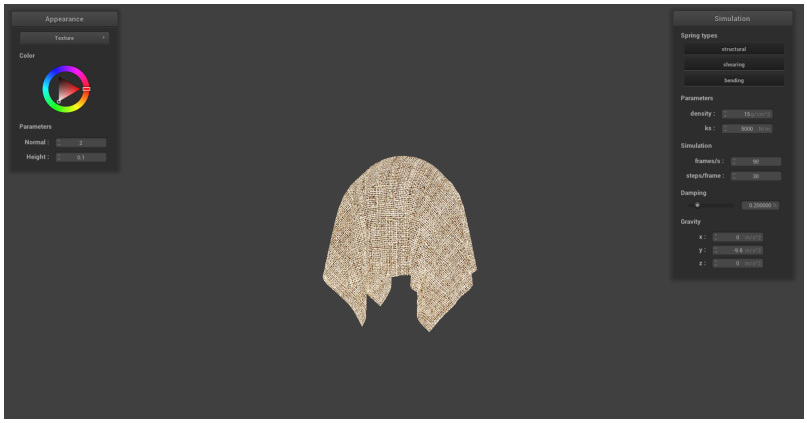
Specular component of Blinn-Phong:



Complete Blinn-Phong lighting:

We also implemented texture mapping as a part our shader implementation. This was done by sampling from a texture at the texture space coordinate uv. We can see an example below with a custom texture mapped to the cloth draped over a sphere.

Texture mapping shader:



Next, we implemented bump and displacement mapping so that our shader has the ability to produce details on an object. For bump mapping, we modified the fragment shader by adjusting the normals of the object so that we can give off the illusions of bumps through shadows. This was done by calculating our new normals in object space and then converting our calculations into model space by using the **tangent-bitangent-normal (TBN) matrix**. The local space normal (converted to object space using TBN) and values for the local normal were calculated using:

$$\mathbf{n}_o = (-dU, -dV, 1)$$
$$dU = (h(u + 1/w, v) - h(u, v)) * k_h * k_n$$
$$dV = (h(u, v + 1/h) - h(u, v)) * k_h * k_n$$

Meanwhile, displacement mapping was

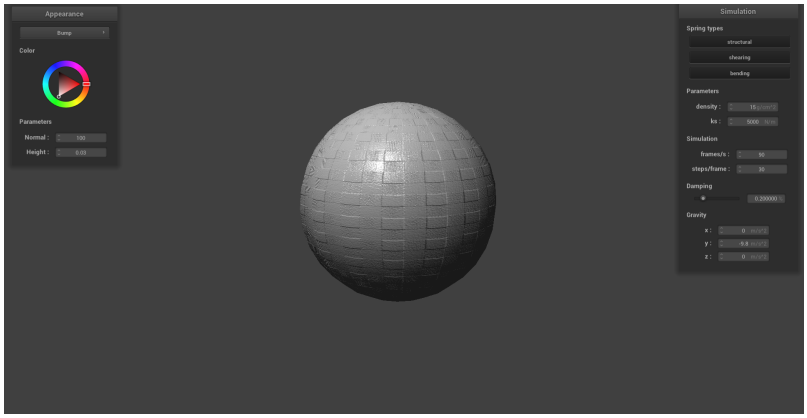
implemented by modifying the vertex shader as well as the fragment shader. In the vertex shader, the positions of the object's vertices were adjusted by physically displacing them in the direction of the original model space vertex normal scaled by the **u_height_scaling** variable. This was implemented by using the following equation:

$$\mathbf{p}' = \mathbf{p} + \mathbf{n} * h(u, v) * k_h$$

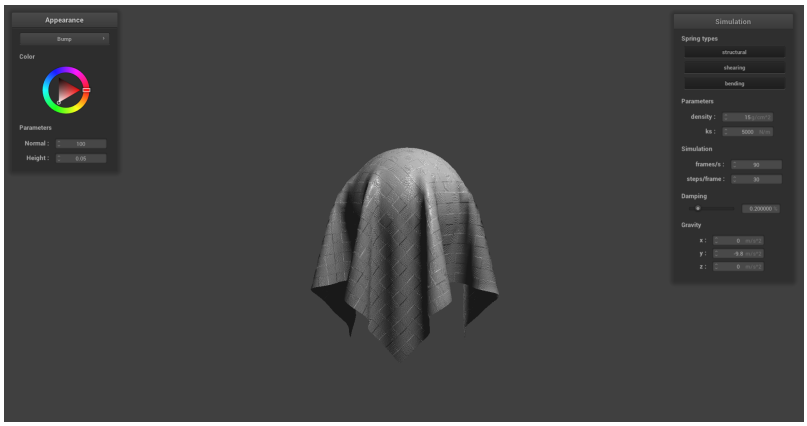
When changing the sphere mesh's coarseness from 16 to 128, we see some very subtle differences in the outputs of the shader. We can see that the sphere in the 128 coarsness image is more jagged for displacement, which leads to more displacement and jaggedness on the cloth. We also see that there are more prominent details in the 128 coarse bump renders as compared to the 16 coarse bump renders.

The height was set at 0.03 while the normal was set to 100 for all of the images generated below.

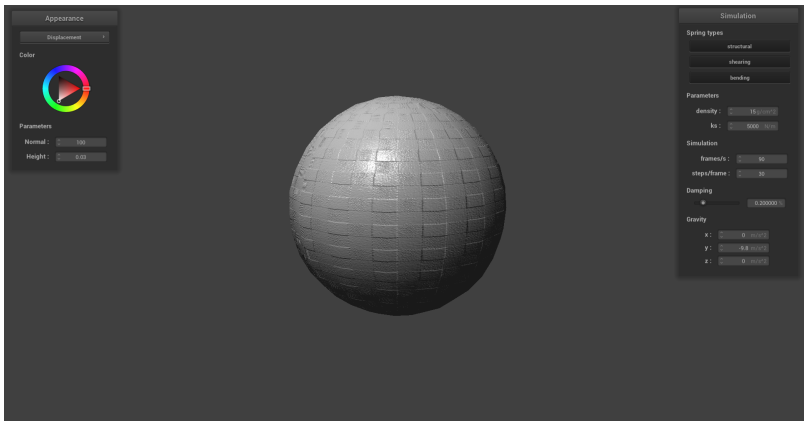
Bump coarsness 16 on sphere:



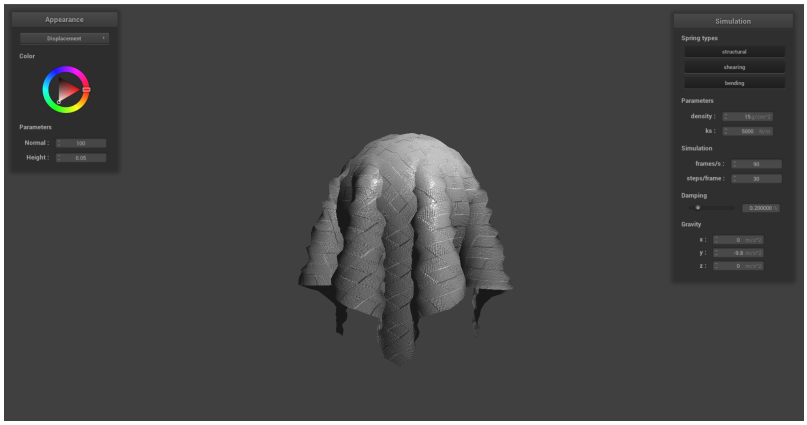
Bump coarsness 16 on cloth:



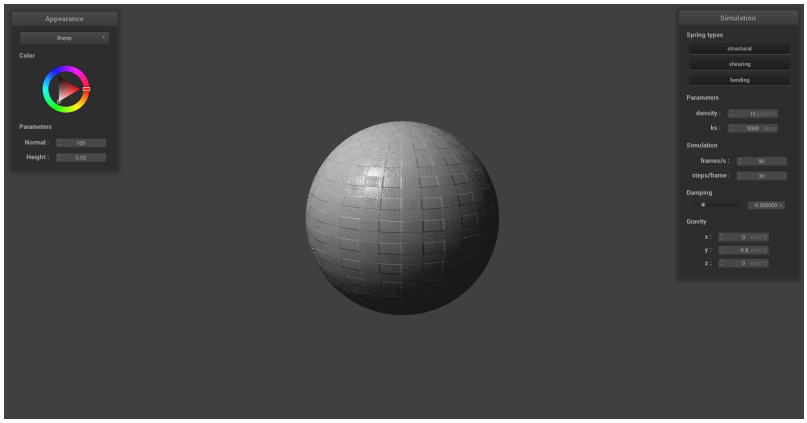
Displacement coarsness 16 on sphere:



Displacement coarsness 16 on cloth:

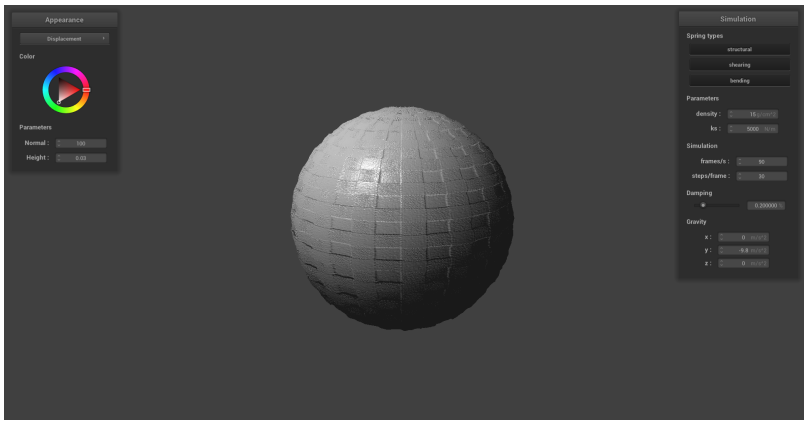


Bump coarsness 128 on sphere:

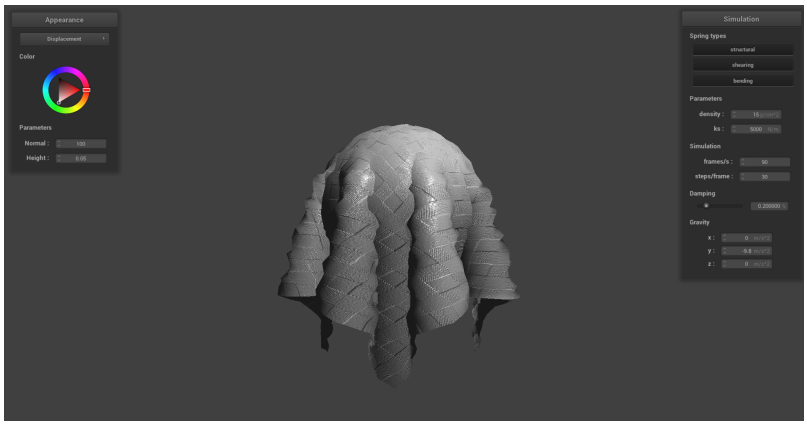


Bump coarsness 128 on cloth:

Displacement coarsness 128 on sphere:



Displacement coarsness 128 on cloth:



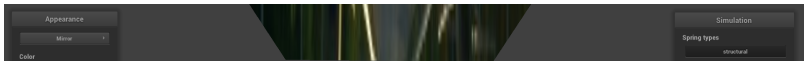
Finally, we implemented a mirror shader that can simulate reflective surfaces on our cloth and sphere as shown below. To implement this, we computed the eye-ray, reflected it across the

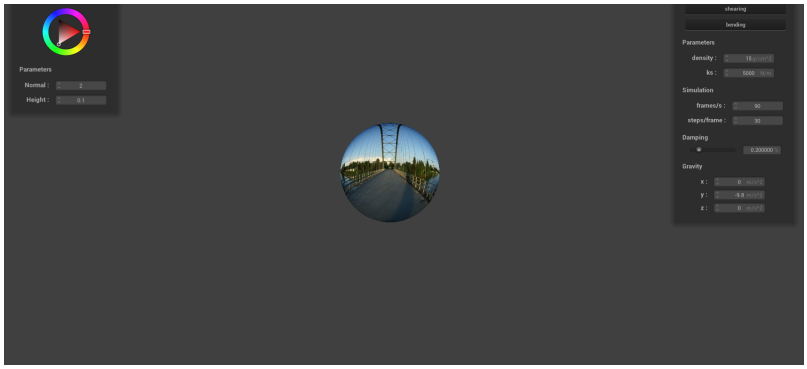
surface normal to get the incoming ray, and sampled the environment map for the incoming direction.

Mirror shader on pinned cloth:

Mirror shader on cloth above sphere:

Mirror shader on sphere:





Mirror shader on cloth draped over sphere: