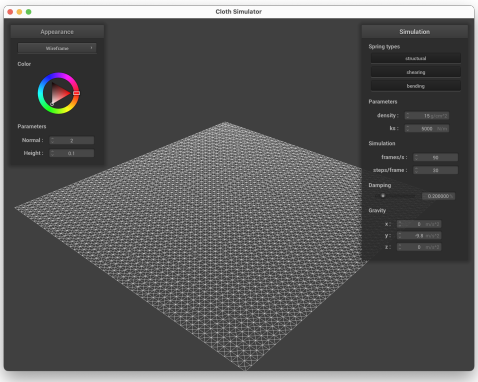
# Assignment 4

## Overview

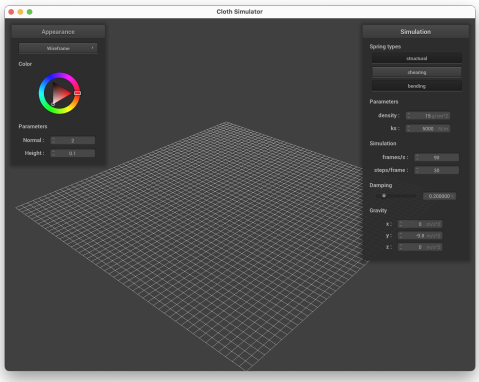
In this project, we implemented a cloth simulater using a mass and spring system. First we built the cloth by creating a grid of point_masses and adding three types of springs types between them. Next, We implemented the simulation step which consisted of accumulating all the external forces and spring forces and applying them to the springs positions. We added the ability to collide with a sphere, a plane, and the cloth itself. Finally, in the last part of this project, we implemented shaders using glsl. We implemented diffuse shading, Blinn-Phong shading, texture shading, bump shading, displacement shading, and mirror shading.

## Part 1: Masses and Springs

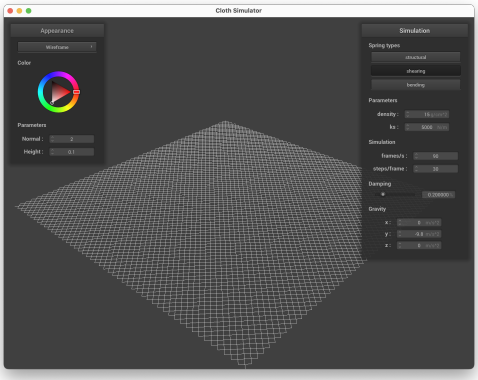
To build the masses and spring, we first set up the grid of masses. We set up the plane space basis vectors based on the `orientation`. Then we start generating the masses. We use the basic vectors to place the massed in a grid. In the special vertical case, we apply the `forward` basis vector randomly to add the random offset. After creating the massed, we iterate through the pinned indices and set the mass `pinned` value to true. To add the springs, we iterate over all the messes we just created. We first create a spring with the mass to the left, and then a spring with the mass above. Similarly, we create springs with the mass two to the left and two above. Finally, the diagonal springs are created. For each direction, we do the correct pounds check before to make sure that there is a mass in that direction.




The wireframe will all constraints displayed




The wireframe with shearing constraints disabled




The wireframe with only shearing constraints enabled

## Part 2: Simulation via numerical integration

First, we sum all the `external_accelerations` and multiply by the mass to get the `external_force`. The forces for every mass is initialized to this force. Next we calculate the spring force by iterating through every enabled spring. We calculate the stretch distance and multiply by the correct `ks` value to get the force. For the mass on each end of the spring, we apply this force in the direction toward the spring center.

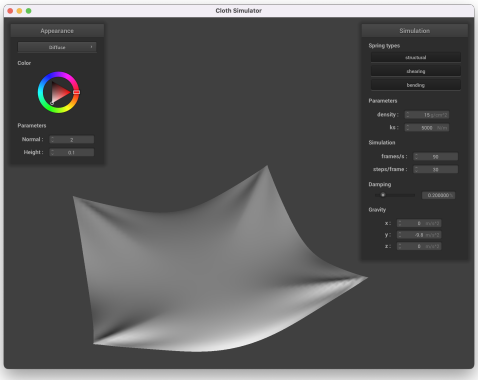
To perform time integration, we iterate over all point masses and save the `position` and `last_position` value in local variables. Then, using the formula from the assignment, we calculate the next position of each point mass. Finally, the `last_position` value is set to the previous `position` value.

We implement the deformation constraints by again iterating through all enabled springs. The spring distance is calculated and check if it is greater than 1.1 times `rest_length`. If it is, we move the springs closer together such that the spring distance is exactly 1.1 times `rest_length`. We handle the cases were neither passed are pinned and the case that only one mass is pinned.

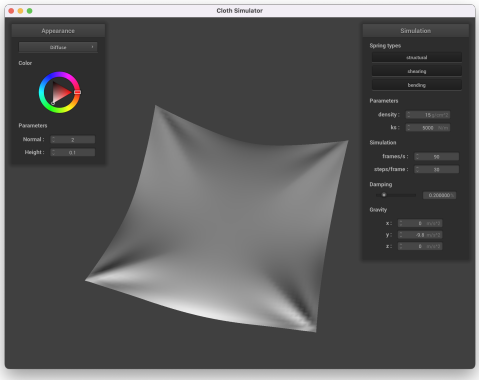
When we lower the value of `ks`, we see that the cloth simulation get more jiggly and less stable. Specifically the part of the close with less tension get very shaky and unstable. As we raise the value, the cloth gets stiffer and more stable. Finally, with a value too high, the cloth gets very unstable and bunches up in a shaky mess.

With the density, when we change make the value smaller the cloth gets somewhere stiff and has few folds develop. The movement of the cloth is very stable like this however, it converges fast and doesn't exhibit any shaky movement. When we increase the density, the cloth gets very stretchy as if moves. The movement gets less stable and more jiggly. When the density becomes very small, the cloth gets unstable and bunches up. This is all inverse to how the simulation changes when we change `ks`. This makes sense, since the density and `ks` value act inversely in the finaly movement calculation.

With a low damping value, the cloth swings and moves a lot. We can see a lot of wrinkles form and disappear and "waves" that move throughout the fabric. When the damping value is increase, the cloth moves much slower and has no oscillatory movement. At the maximum value, the cloth slowly moves into position and doesn't oscillate at all.




pinned4.json in its final resting position




pinned4.json in its final resting position

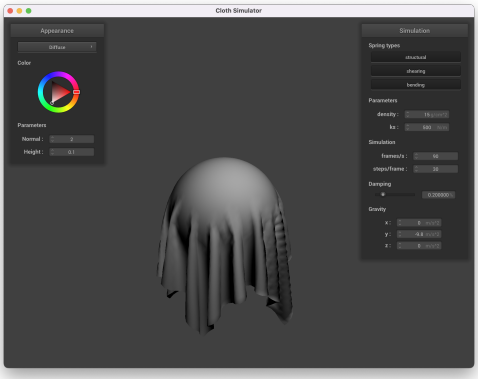
# Part 3: Handling collisions with spheres

## Sphere Collision

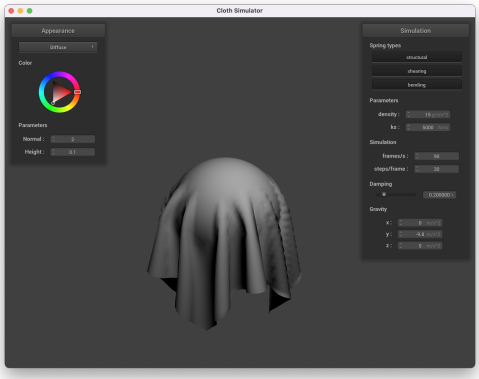
Handling sphere collision is fairly straightforward. First we check if the point is inside the sphere by simply checking if its distance from the sphere origin is less than the radius. If it is, we calculate the tagent position by extending a vector from the sphere origin, passing through the point position, and to the sphere surface. The correction is just the difference between the tangent position and the `last_position`. This correction is applied accounting for the `(1 - friction)` coefficient.

## Plane Collision

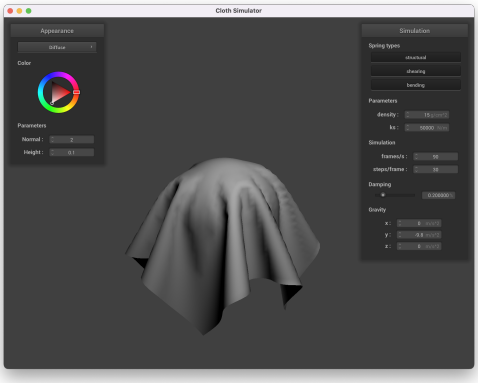
To handle plane collision, we act as if we cast a ray from `last_position` to `position`. We calculate this the `t` value using the formula given in the ray casting lectures. If the collision point is between the two position values, then a collision occurred. The tangent point is just the collision point from the ray cast. The correction is once again the difference between the tangent position and the `last_position`. However, this time we add `sign * SURFACE_OFFSET * normal` to the correction, where `sign` is the sign of `dot(pm.last_position - point, normal)`. The correction is applied the same way as before.




The sphere resting position with `ks = 500`




The sphere resting position with `ks = 5000`




The sphere resting position with `ks = 50000`

We see from these images that with low values of `ks` the cloth is more flexible and behaving almost like silky fabric. The cloth conforms to the shape of the sphere more and ends sag down lower. The folds are sharper and denser. As the value is increased, there are less folds. At `ks = 50000` we see that the cloth appears stiff near the edges and doesn't hang nearly as much as the other two.

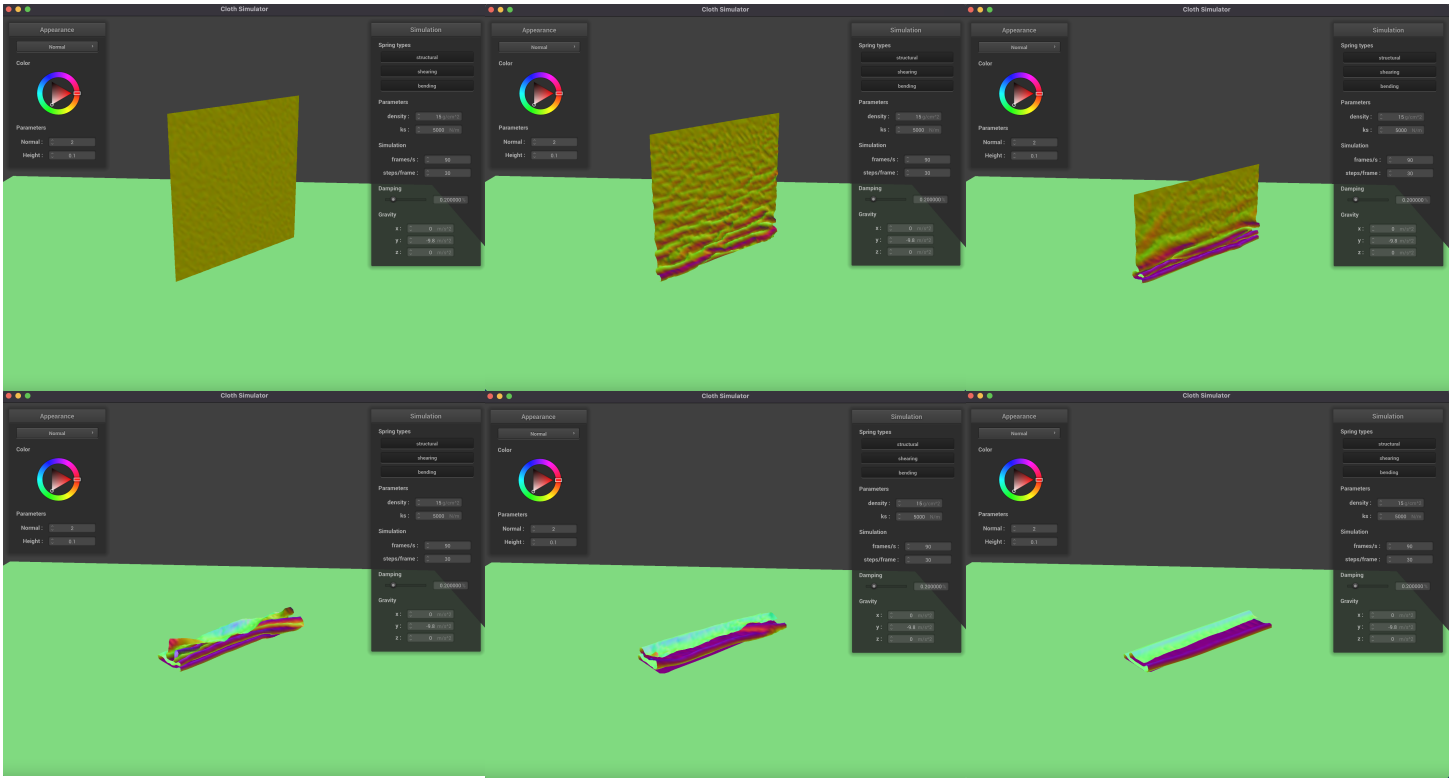
# Part 4: Handling self-collisions

For this part in the project we need to program self-collisions because otherwise our cloth just falls through itself. The naive solution would be for every single point mass, loop through every other point mass and check the distances between them, but this would take O(N^2) time per call to simulate, which would be too slow.

Instead we section the world into boxes and then store a list of point masses per box, similar to what we did for BVHs. To access each box as fast as possible we would store each point in a hash map with a unique key representing which box it belonged to, allowing us O(1) access to the list of points in a box. For this we implemented `Cloth::hash_position` that given a position would return the unique box id. We accomplished this by taking the floor of the division of the position's x value by the width of a box, and doing the same for the position's y value and the height, and the position's z value and the max of the box's height and width. These three points effectivly gave us the top left corner of which box the point lies in, to convert this into a unique id we just concatenated the x, y, and z values.

Finally, we implemented `Cloth::self_collide` that given a point mass would check if it would collide with other points on the cloth and if so correct the position to avoid such collision. To implement this we would get the id of the box the point mass lies in and then check that its distance with every other point mass in the box was less than `2 * thickness`. If any points were too close we would sum a correction vector to move the point farther than `2 * thickness`. At the end we would average this correction vector and move the given point mass.

Once we had all these functions implemented, we populated the map of boxes to list of point masses at each step in the simulation, and checked for any self collisions with the points.



Varying the density:

| cloth falling d_1 | cloth falling d_30 | cloth falling d 500 |
| --- | --- | --- |
| Density = 1 | Density = 30 | Density = 500 |

When adjusting the density values we see that when the density is low, the cloth acts more paperlike with bigger folds over itself. When the density is high the cloth acts more silklike folding over itself with small folds. When the density gets to high, we see that the material starts to phase through the plane. This is probably caused by a large number of really tight folds so each point mass experiances really large repulsion forces that push it through the plane.

Varying the ks value:

| cloth falling k 2500 | cloth falling k 7500 | cloth falling k 10000 |
| --- | --- | --- |
| ks = 2500 | ks = 7500 | ks = 10000 |

We can see the lower Ks values show a much springier cloth material that bounces a lot before reaching its final rest position. This cloth bounces so much that it actually unfolds most of itself by the final position. While at higher ks values the cloth is stiffer and does not bounce as much. This also means that if it is unfolding itself it takes a longer time to reach its final rest position.

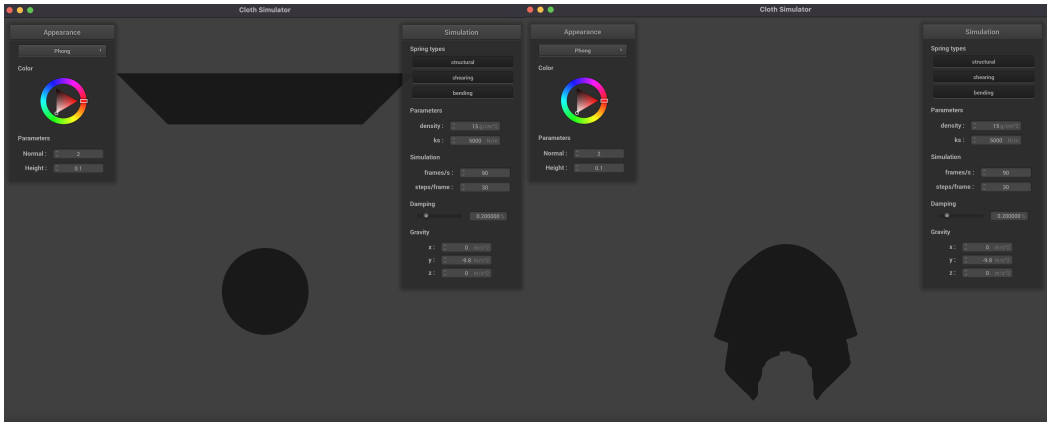
# Part 5: Shaders

For the final part, we implemented shaders using GLSL. Shaders are incredibly powerful because they run many computations in parallel on the GPU. They take in atributes, which are inputs into the program, like position, texture files or uv coordinates, and writes to varyings, which are like the outputs of the program, such as the color or displacement at the vector or pixel. In OpenGL there are two types of shaders, there are vertex shaders and fragment shaders. Vertex shaders apply transformations to the vertices, such as displacement. While, fragment shaders write to the attributes on the pixels, which are created after rasterization.

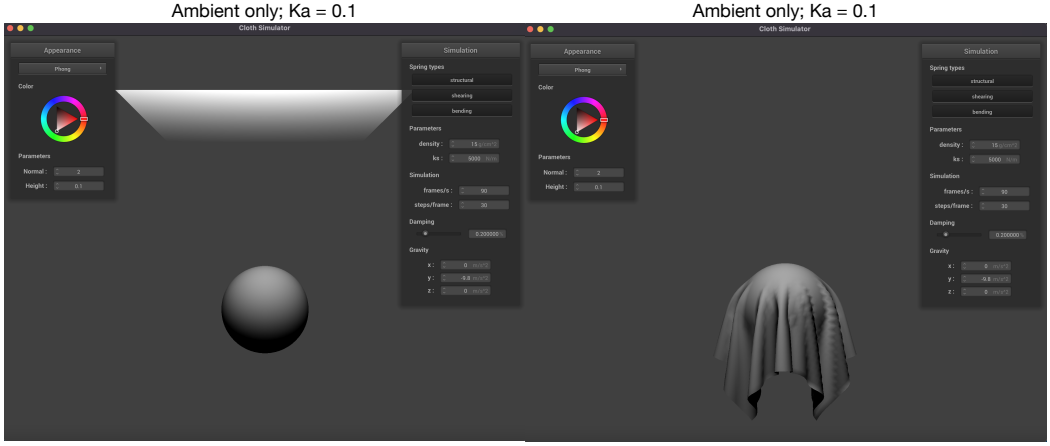
In the Blinn-Phong shading model, we can combine ambient lighting, diffuse lighting, and specular highlights. Each of these components is represented by its term in the lighting calculation:

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

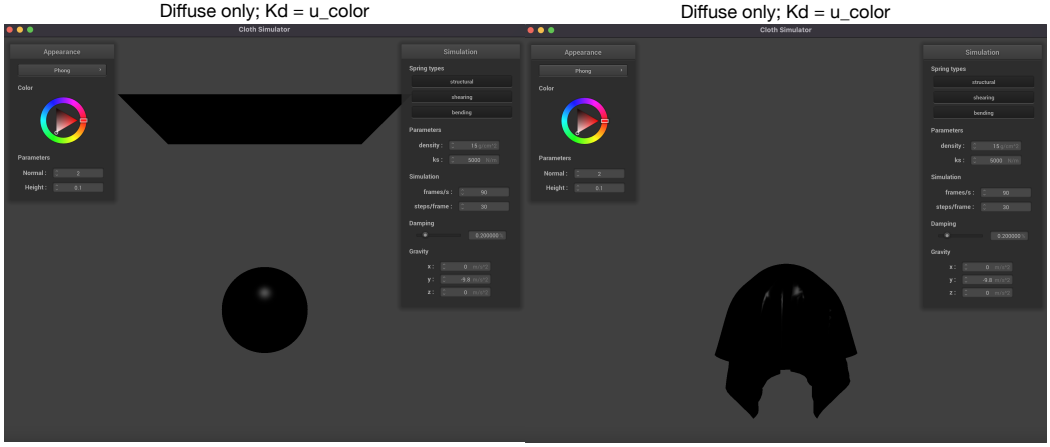
Blinn-Phong shading parts:



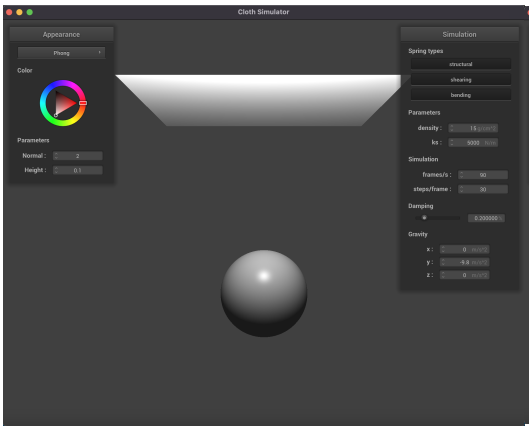
Ambient only; Ka = 0.1 | Ambient only; Ka = 0.1



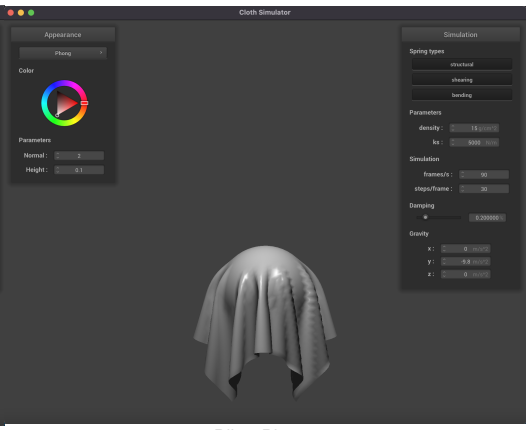
Diffuse only; Kd = u_color | Diffuse only; Kd = u_color



Specular only; Ks = 0.5 | Specular only; Ks = 0.5

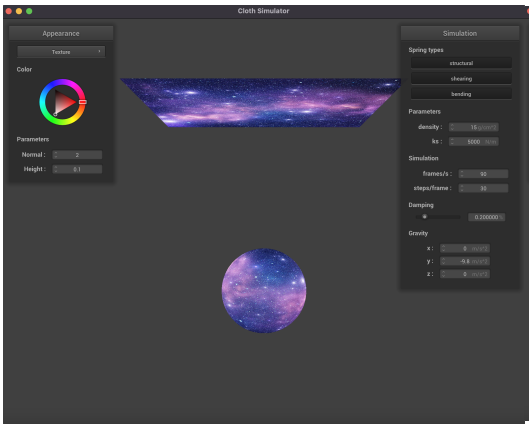


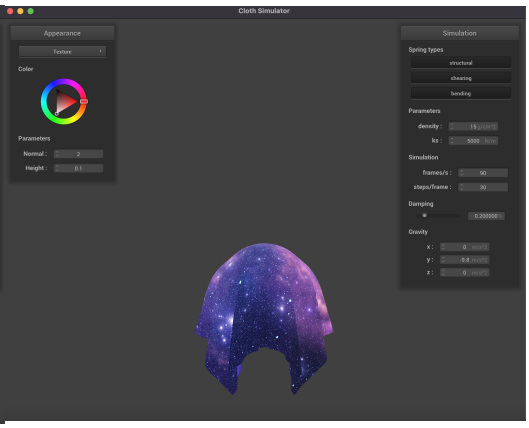

Blinn-Phong



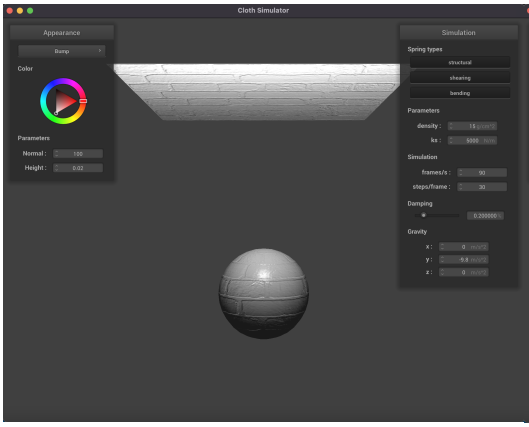

Blinn-Phong

Texture Mapping Shader:

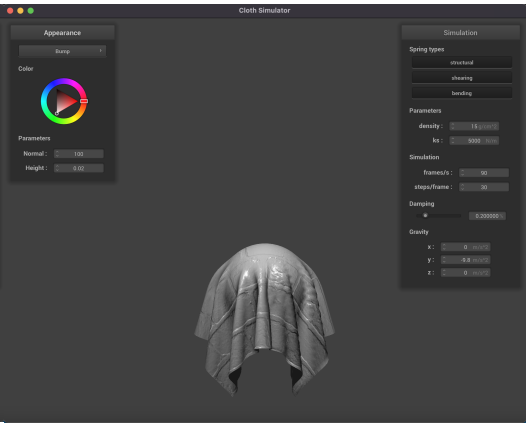


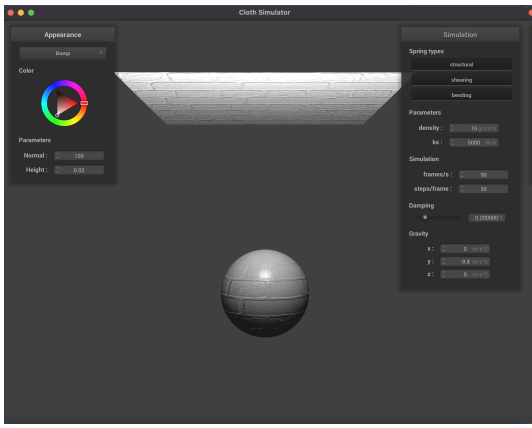

Bump Shader and Displacement Shader:



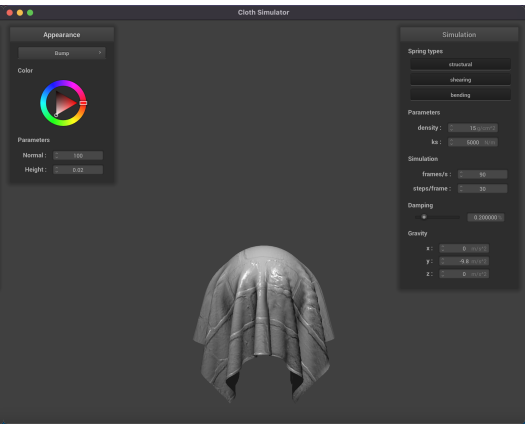

Bump, coarseness 16



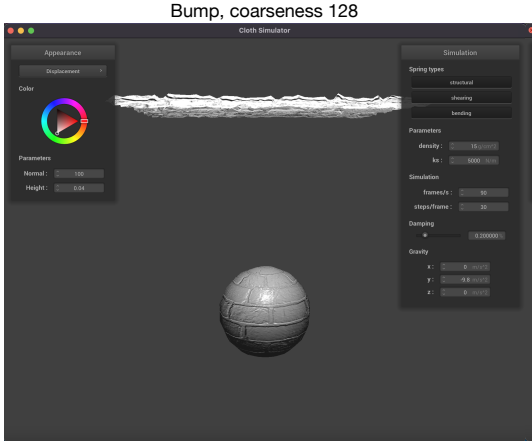

Bump, coarseness 16



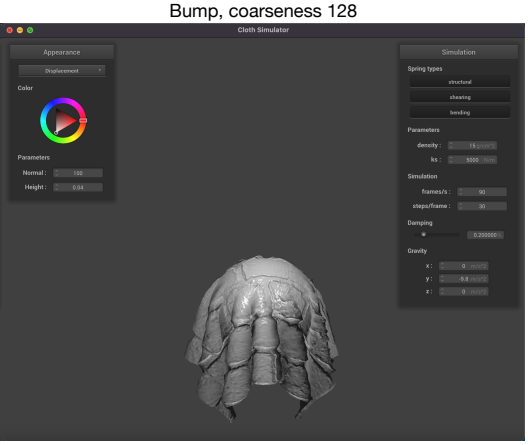

Bump, coarseness 128




Bump, coarseness 128

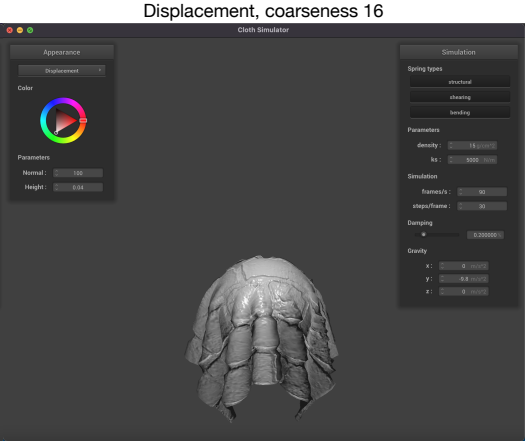



Displacement, coarseness 16




Displacement, coarseness 16

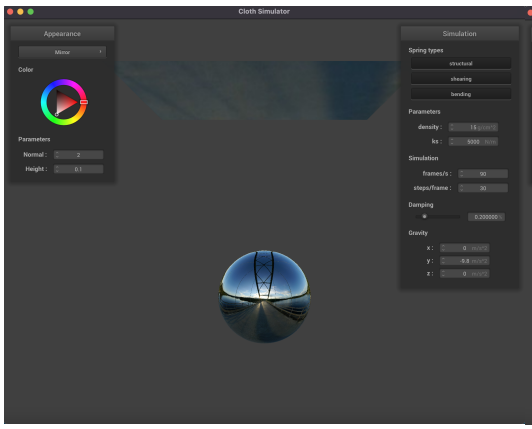

Displacement, coarseness 128

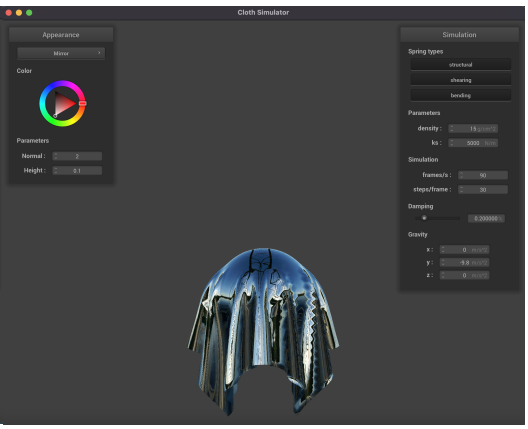



Displacement, coarseness 128

We can see that at a coarseness of 16 the difference between bump and displacement shading is not as pronounced as when we up the courseness to 128, where we can see a greater difference between bump and displacement shading, which makes sense as the courser the texture is, the more displacement we expect from the shader.

Mirror Shader:



https://cal-cs184-student.github.io/sp22-project-webpages-AdamRashid96/proj4/index.html (https://cal-cs184-student.github.io/sp22-project-webpages-AdamRashid96/proj4/index.html)