CS184: Computer Graphics and Imaging, Spring 2024

Project 4: Cloth Sim

Pranav Kolluri, Rohan Agrawal

Introduction

In this project, we implemented a cloth renderer using a mass-spring system. We started by creating a 2D grid of point masses, and then added springs between them to simulate the cloth. We then implemented numerical integration to simulate the motion of the cloth. We then added collision detection with other objects, namely with a sphere and a plane. Finally, we implemented self-collision detection to simulate the real world behavior of cloth in a more realistic manner.

Part 1: Here we implemented the basics of cloth simulation. We created the base, a system of masses connected to springs that would simulate structure, as well as shearing and bending forces.
Part 2: Next we implemented the actual physics that made that system of weights and balances behave like a cloth. We used Hooke's law, as well as Verlet integration to simulate how forces acting on the masses would make them behave, especially in relation to other masses they are connected to with different types of springs.
Part 3: In this part we moved on to simulating how the cloth would interact with the objects in the world space. We primarily did this by calculating its position, then adjusting each point mass in relation to where it should be to the sphere or plane it interacted with.
Part 4: The last step of simulating interaction with the world was making the cloth interact with itself, to prevent it from collapsing into a physical impossibly shape. To do this we employed a hash table and corrected the position of each point based on other points that were close enough to fall into its hash box.
Part 5:

1. We first implemented shaders, an instruction set that allows the GPU to render graphics. This involved implementing vertex and fragment shaders, which work together to create realistic lighting and material effects.
2. We then implemented Blinn-Phong shading, which is an extension of the Blinn-Phong reflection model, which takes Ambient, Diffuse, and Specular reflection into account to calculate the color of a pixel on the surface.
3. Lastly we implemented Bump and Displacement mapping, which uses shading combined with preprocessing step that recalculates the normals of the object based on a texture. This

allows for the appearance of more detail on the object without actually changing the geometry of the object

Each part of the project builds upon the previous one, gradually enhancing our understanding of cloth rendering and simulation.

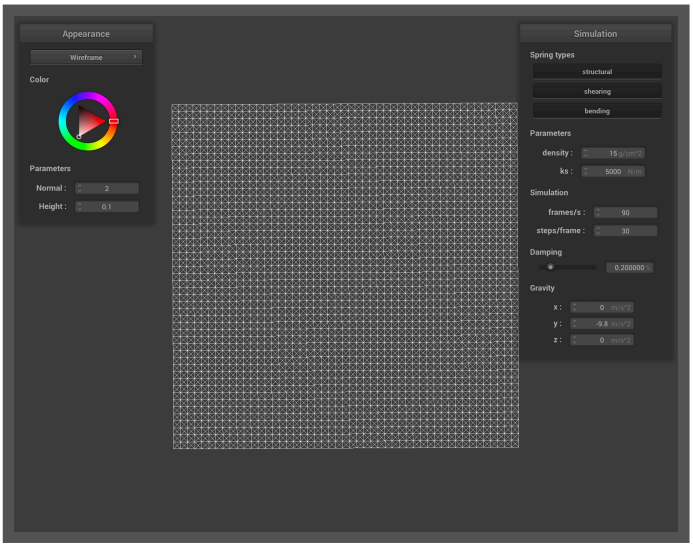
Part 1: Masses and springs

Our mass-spring system works as follows:

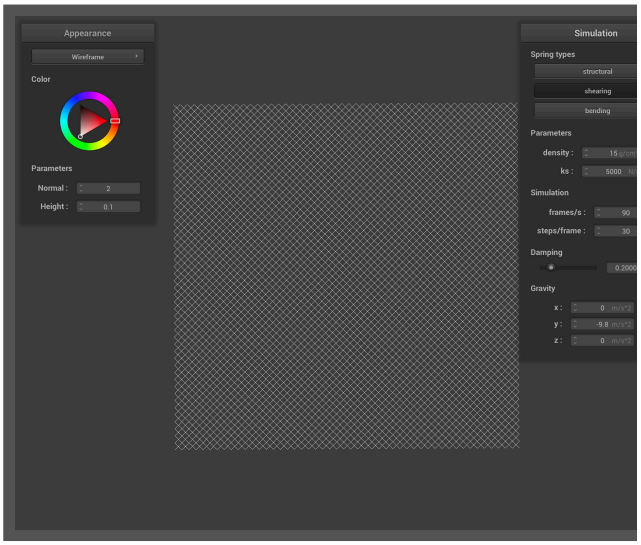
1. We loop over the 2D grid that is height points tall and width points tall.
2. At each point we check if it is horizontal or vertical. If it is vertical we add a point that that location, with the y set to 1
3. If the orientation if Horizontal, we add a point at that XY coordinate, and vary generate a random small offset for the Z coordinate

Next we add the springs between the points. We add springs between the points in the following order:

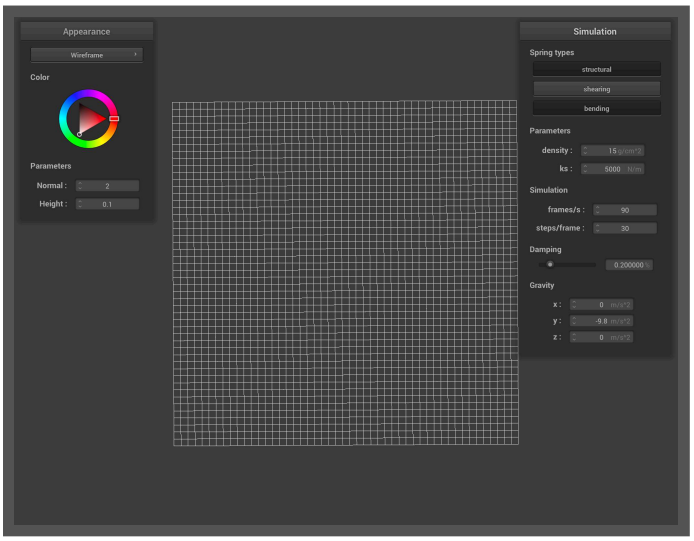
1. Horizontal Springs: We add springs between the current point and the point to the right of it
2. Vertical Springs: We add springs between the current point and the point below it
3. Diagonal Springs: We add springs between the current point and the point to the right and below it
4. Anti-Diagonal Springs: We add springs between the current point and the point to the right and above it



The wireframe with all constraints!



The wireframe with only the shearing constraints!

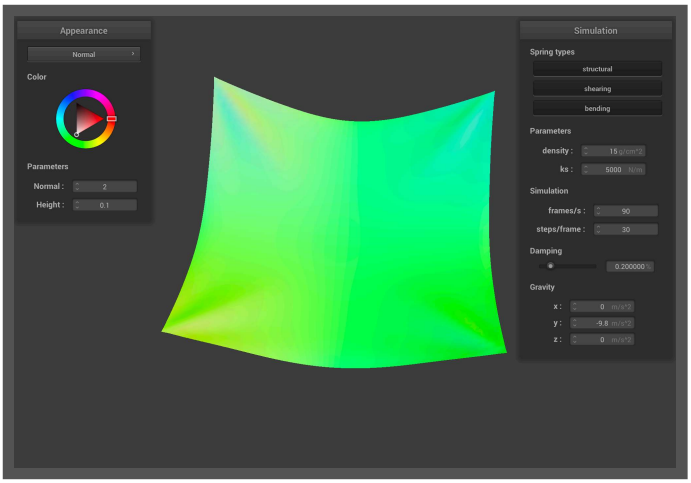


The wireframe without the shearing constraints!

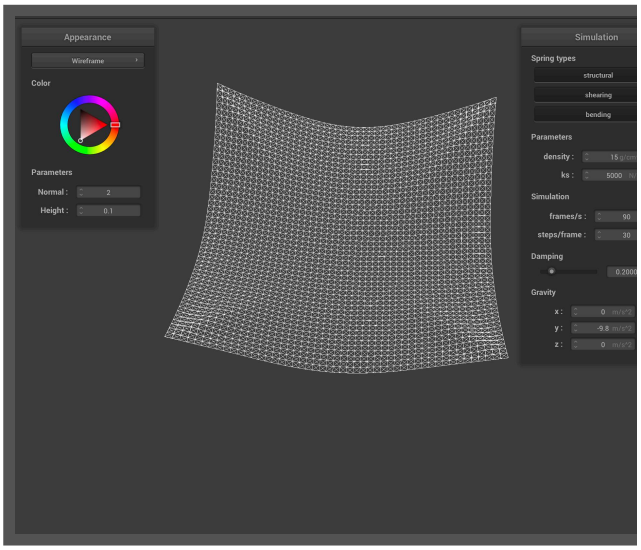
Part 2: Simulation via numerical integration

To simulate the cloth we use various physics equations to calculate the forces acting on the point masses. We then use numerical integration to calculate the new position of the point masses.
Our simulation works as follows:

1. We iterate over every spring and determine if its constrain type is enabled. If it is we model is physically, otherwise we skip it.
2. For each spring we calculate the force acting on the spring using Hooke's law, which is F_s = k_s * (||p_a-p_b||-l), where k_s is the spring constant, l is the spring's rest length, and p_a and p_b are the positions of the point masses.
3. Next we use Verlet integration to compute new point mass positions. We use the equation x_(t+dt) = x_t + (1-d)*(x_t-x_t-dt) + a_t *dt^2 to calculate the new position of the spring, where d represents a damping factor, and dt is a timestamp delta_t.
4. Lastly we adjust the springs update so that it is not more than 10% greater than its rest length to prevent unreasonable deformation.



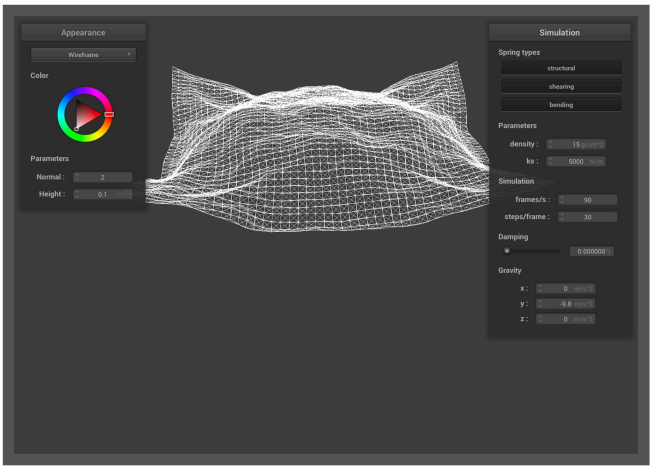
Cloth at rest



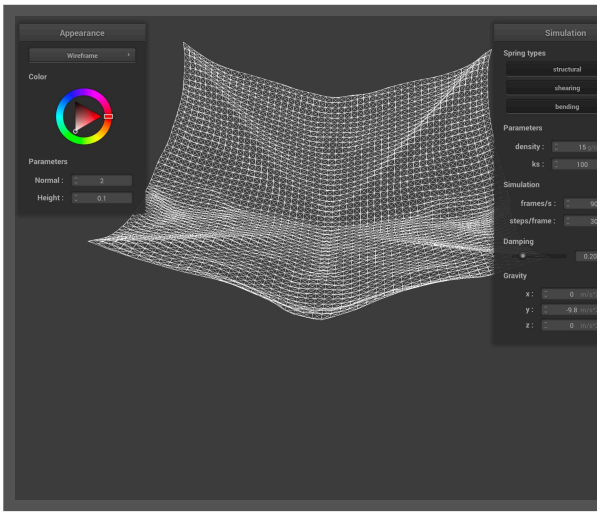
Wiremesh at rest!

Observations:

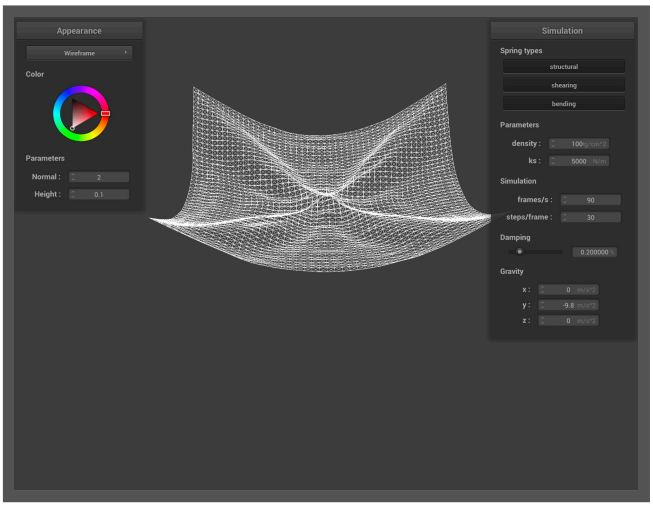
1. As the density decreases the cloth comes to a rest quicker, likely because there is less energy stored in the system, as opposed to a higher density, which results in a bounce back that is less observable when the density is lower.
2. As the cloth density decreases it bounces back further each time
3. As damping increases the body moves slower and bounces less at the bottom, coming to a restful state quicker. This is contrasted with less damping, which allows the springs to maintain their energy longer, allowing them to bounce for longer periods of time.
4. As the spring constant was decreased, the bounce increased, however this flattened after a certain point due to the 10% stretch limit we imposed. As the spring constant increased the wireframe became more stiff.



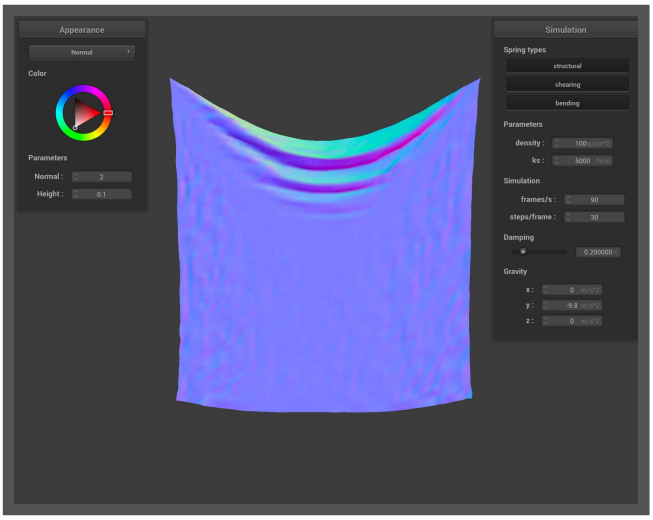
The springs bouncing back after hitting the bottom with zero damping



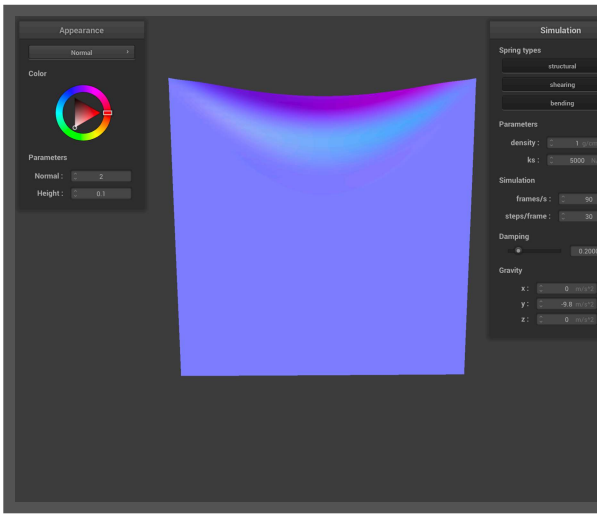
Increased bounce after decreasing the spring constant to 100!



The mesh boucing back after increasing the density to 100!

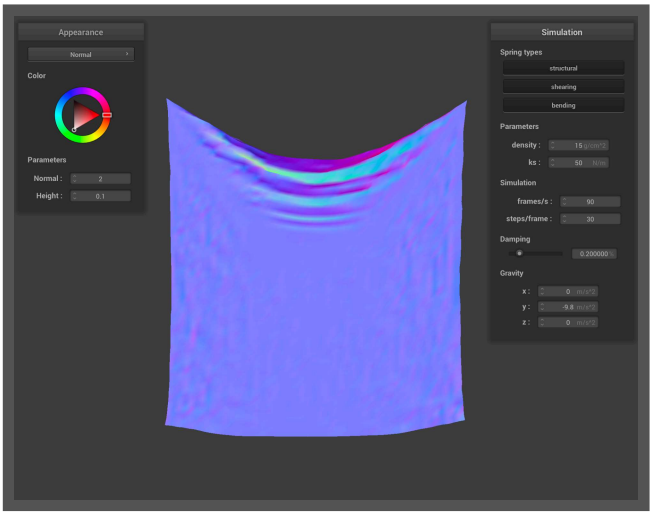


The cloth with a very high density (1000)

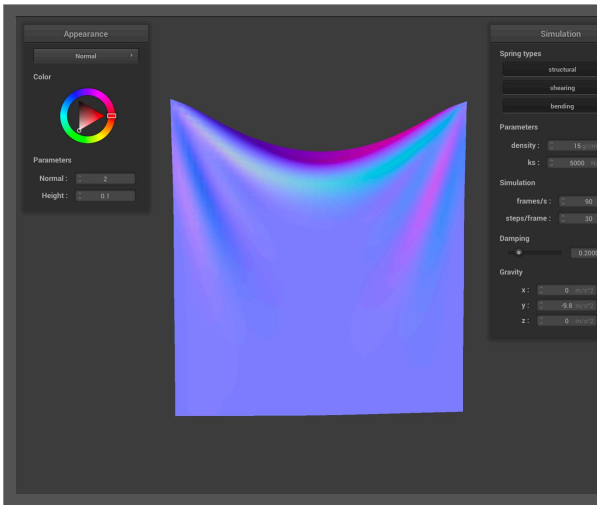


The cloth with a very low density (1)!

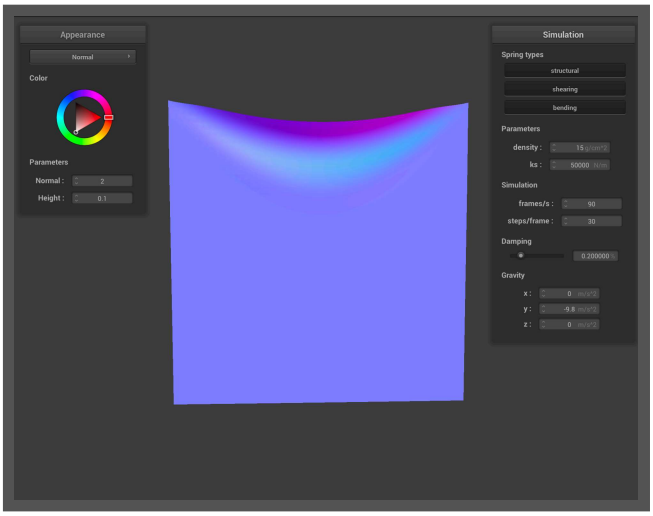
From this we can see the effects of the increased density - the cloth has a desire to hang more, as could be expected in the real world from a more dense cloth, as it would weight more and be more prone to the effects of gravity. The converse is true of the less dense cloth, and it hardly pulls on the mounting points at all.



The cloth with a spring constant of 500



The cloth with a spring constant of 5000



The cloth with a spring constant of 50,000!

Here we can see the effects of changing the spring constant, as we decrease the spring constant the Cloth appears to have a lighter appearance, more prone to the affects of gravity, and this hanging more at the top. This can be contrasted to the higher spring constant, which appears to be more stiff, and hangs very little from the mounting points. Curiously, the spring constant visually seems to correlate inversely with the effects of density.

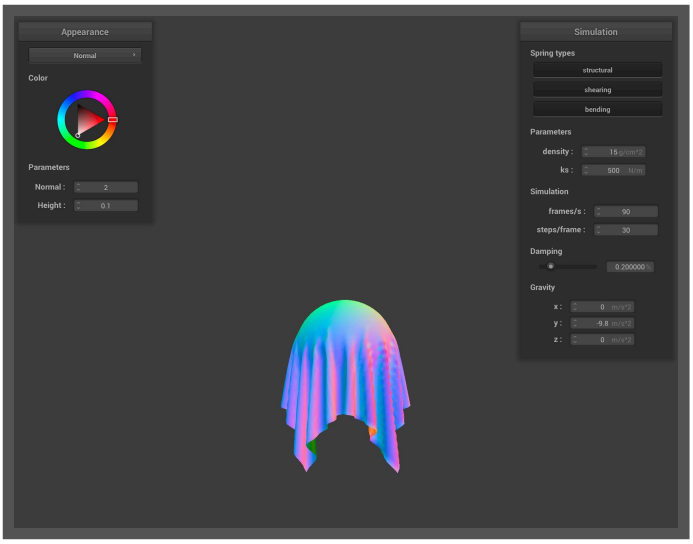
Part 3: Handling collisions with other objects

In part 3 we implemented interaction with other objects, namely the sphere and the plane. These two interactions are implemented in a fairly similar method for both, first we check if there is an intersection between a pointmass, then calculate where it needs to move to not intersect, then adjust the location of the pointmass.

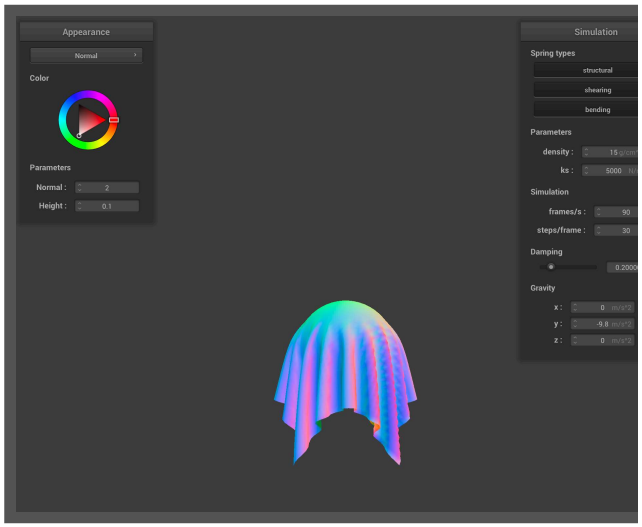
Our sphere intersection algorithm works as follows

1. First we check if the pointmass is inside the sphere by checking if the norm of the point mass minus the origin of the where is less than the radius
2. If it is, we calculate an adjustment vector by subtracting the position of the point mass from the radius times the origin in the direction of the point mass
3. Finally we scale this value by 1-friction, and add it to the point mass

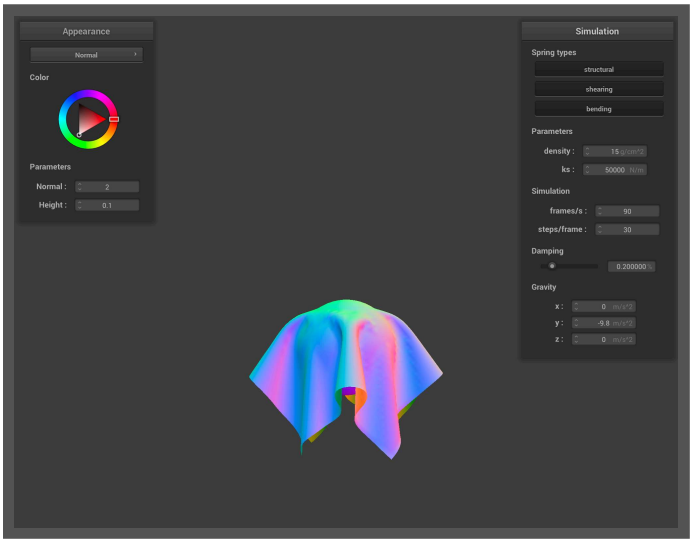
Some different views of the Sphere, with varying spring constants



The spheres with a spring constant of 500



The spheres with a spring constant of 500

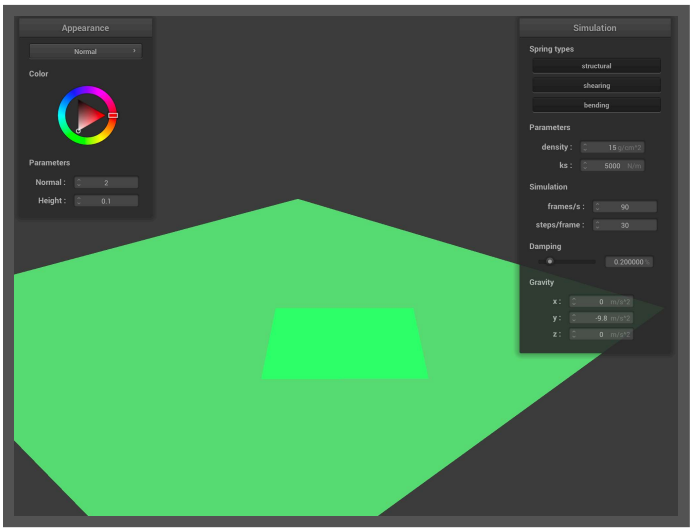


The spheres with a spring constant of 50,000

Here we can see the how increasing spring constant affects the cloth's interaction with the sphere. As the spring constant increases, the cloth becomes more stiff, and as such we observe that the cloth becoming less prone to deformation and to small wrinkles. The cloth effectively reduces the amount of draping that occurs, with the cloth appearing to effectively hold itself up.

Our plane intersection algorithm works as follows

1. First we check if the pointmass is inside the plane by checking if the dot product of the
2. If it is, we calculate an adjustment vector by subtracting the position of the point mass from the radius times the origin in the direction of the point mass
3. Finally we scale this value by 1-friction, and add it to the point mass



The cloth on the plane

Part 4: Handling self-collisions

Self collisions relies heavily on the hash map data structure. It works by storing the point masses in a hash map, and then checking each point masses hash and checking if it contains other point masses. If it does, we calculate the adjustment vector and add it to the point mass to prevent collisions.
Our hashing algorithm works as follows:

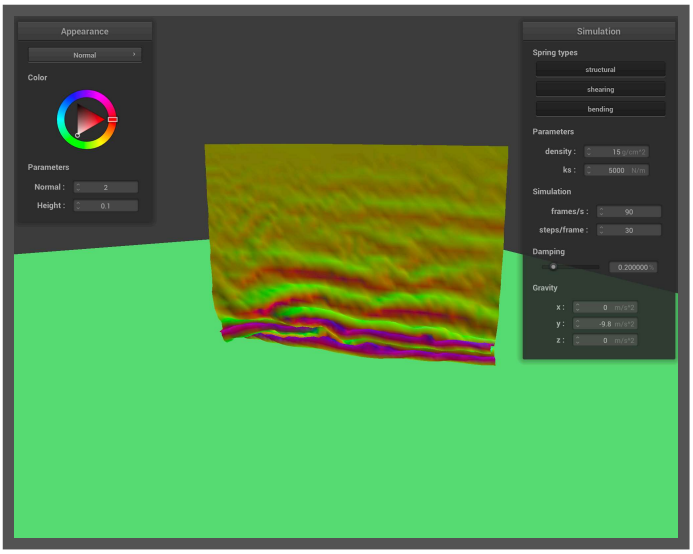
1. We calculate three variables, w, h, and t.
2. w is assigned to 3 * the width of the cloth divided by the number of width points
3. h is assigned to 3 * the height of the cloth divided by the number of height points
4. t is max of w and h
5. We then check if the orientation is horizontal or vertical. If it is horizontal we floor divide the position of the point mass (x, y, z) by (w, h, t), respectively, and add and return them.
6. If it is vertical, we floor divide the position of the point mass (x, y, z) by (w, t, h), respectively, and add and return them.
7. We then return that value we calculated, and that serves as the hash of the point.

After that we create a hash table of all the point masses. To detect collisions we used an algorithm that worked as follows:

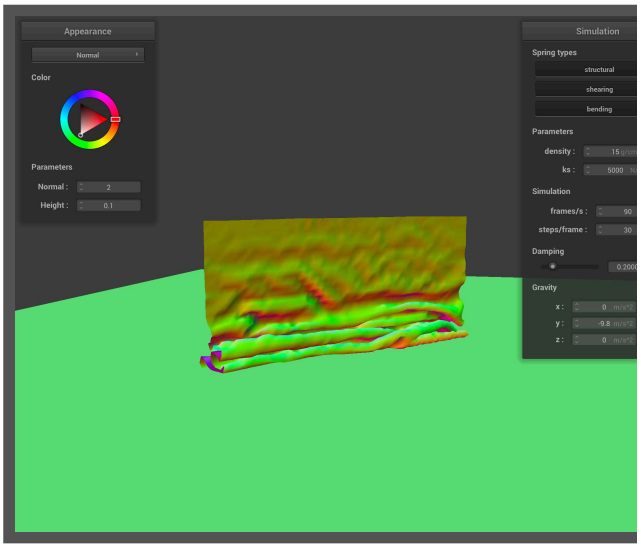
1. We get the hash of the point mass and check if it has any other points in its hash map box.
2. If it does, we calculate an adjustment vector by creating a vector when added to the point masses position will push it to be 2 * thickness units apart.
3. We then add this to the total correction vector

4. Finally we divide the correction vector by the number of corrections made, times the simulation steps, and add it to the point mass.

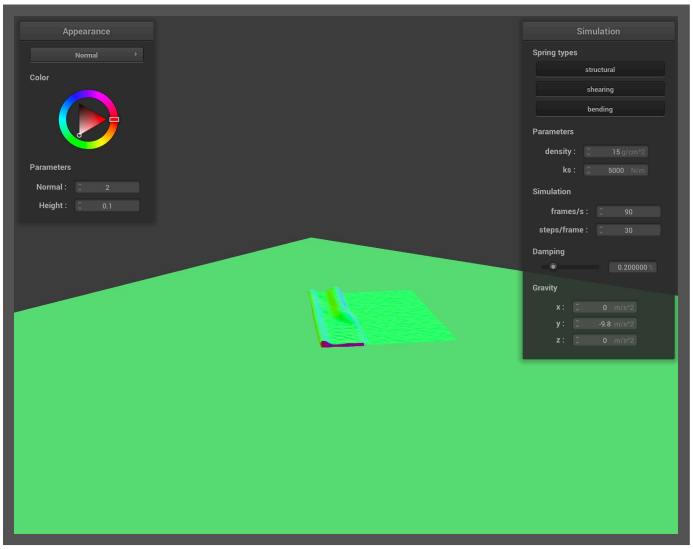
Let's take a look at the cloth rendered with different densities and spring constants:



Density 15 ks 5000 Cloth falling

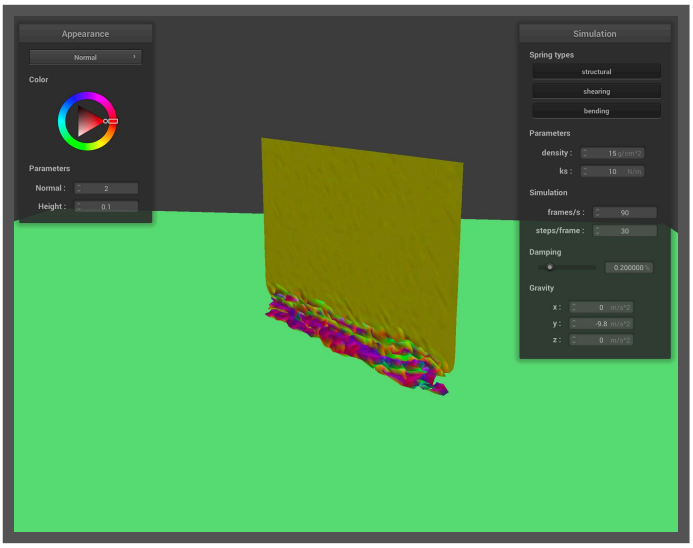


Density 15 ks 5000 Cloth at intial self collisi

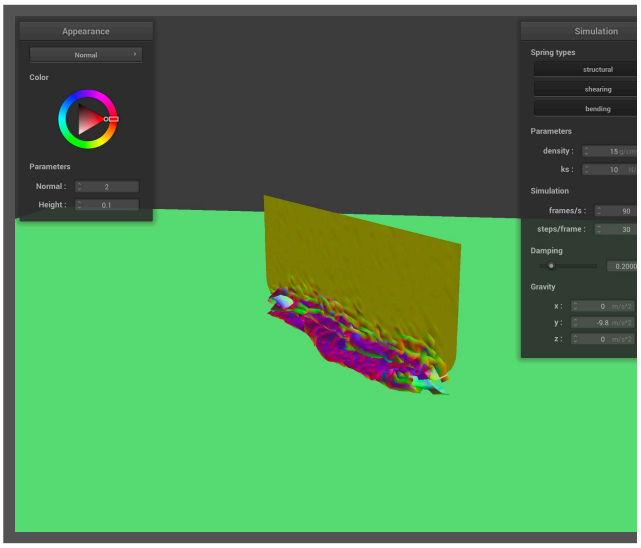


Density 15 ks 5000 Cloth at rest

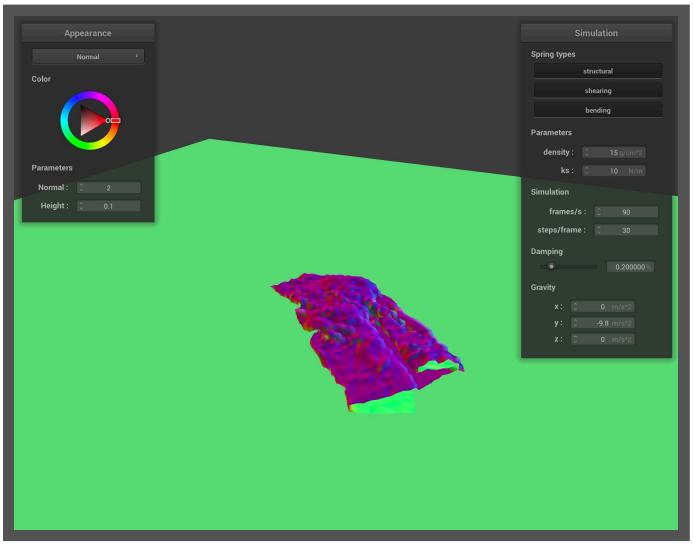
Experiments with Spring Constants



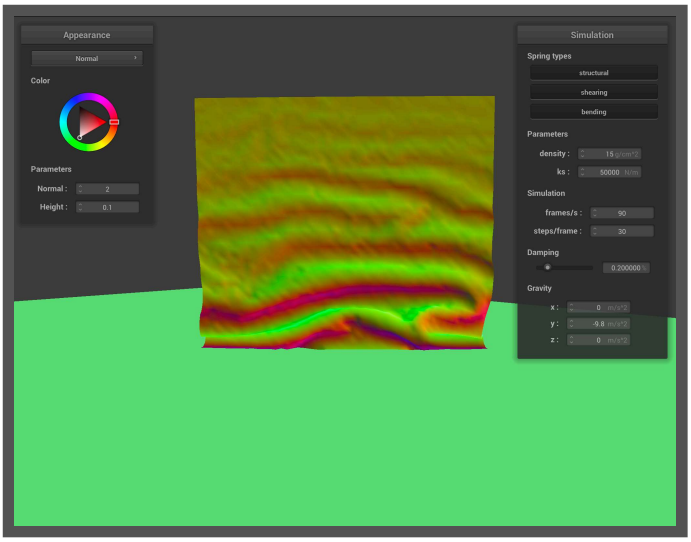
15 Density 10 ks Cloth at start of fall



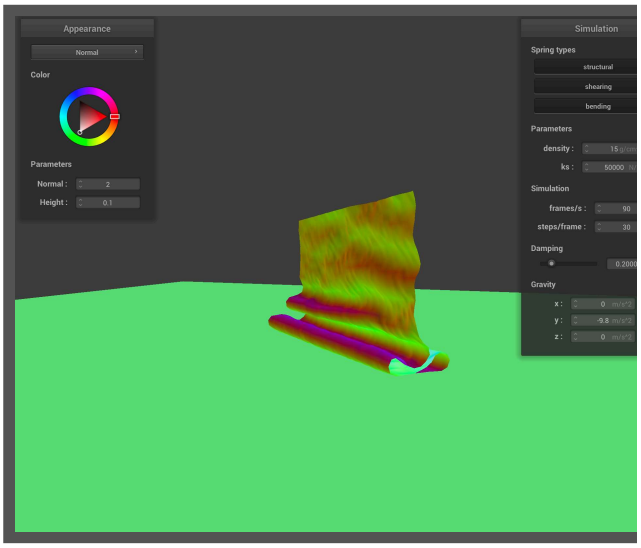
15 Density 10 ks Cloth at middle of fall



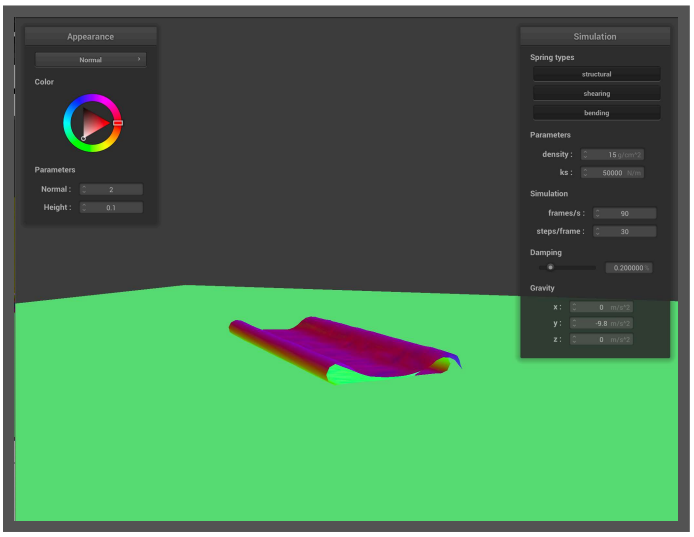
15 Density 10 ks Cloth at rest



15 Density 50000 ks Cloth at start of fall



15 Density 50000 ks Cloth at middle of fal

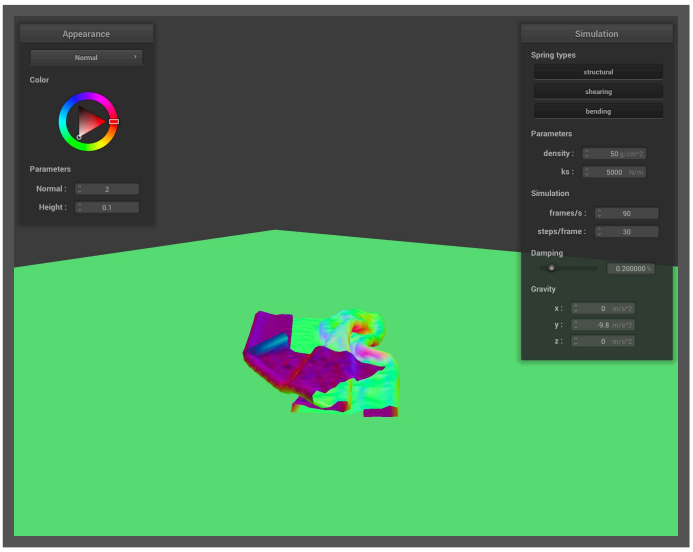


15 Density 50000 ks Cloth at rest

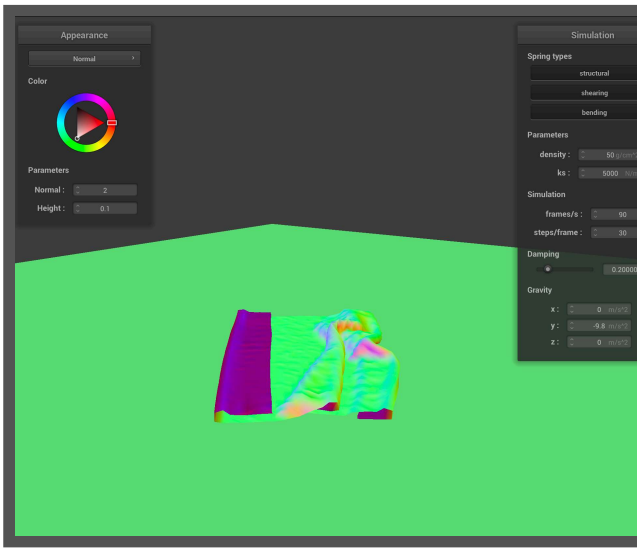
Effects of changed spring constant:
As we can see, the higher spring constant cloth contains only a few large folds, likely due to the stiffness of the springs with forced the cloth to stop moving quicker. This is contrasted with the lower spring constant cloths, which came to rest after a longer time, and formed lots of folds. The higher spring constant gives the cloth a tighter woven, more stiff appearance, while the lower

spring constant gives the cloth a more loose, flowing appearance. The lower spring constant cloth results in folding that is more chaotic and more fluid-like!

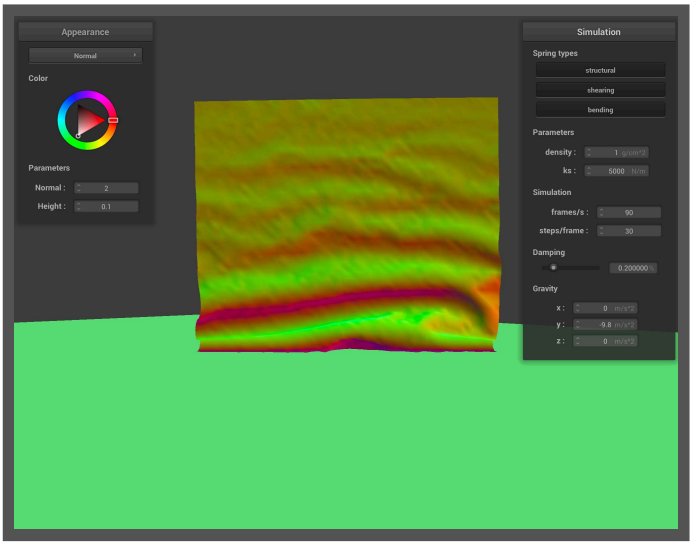
Experiments with density



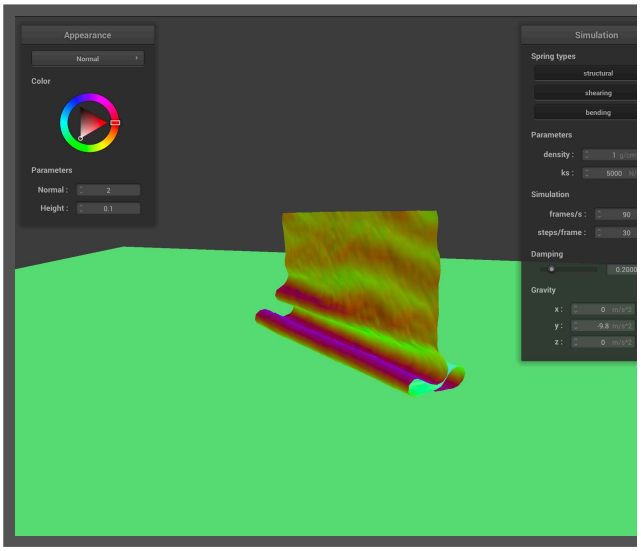
Density 50 ks 5000 Cloth at crumpling



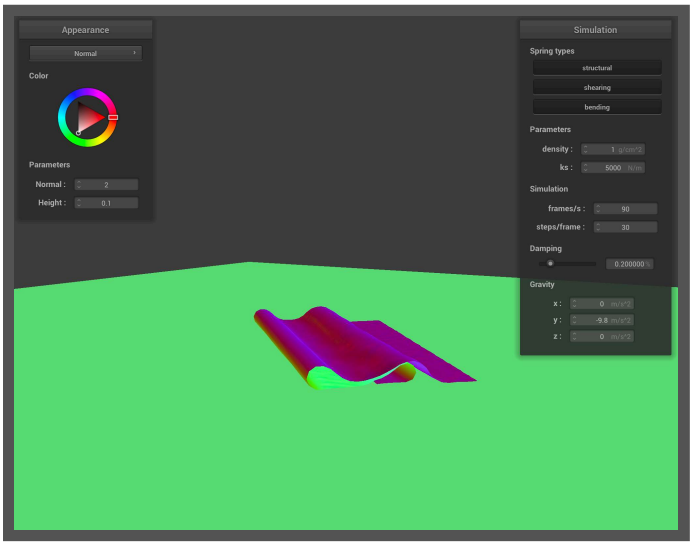
Density 50 ks 5000 Cloth at rest



Density 1 ks 5000 Cloth at crumpling



Density 1 ks 5000 Cloth at rest



1 Density 10 ks Cloth at rest

Effects of changed density:
As we can see, the higher density cloth seemed to behave as can be expected of an object with more density, having come to a rest quicker, and with more folds remaining at the restful state. The reduced density appears to fold neater, likely as the small perturbations in the cloth are less pronounced in the folding physics due to the reduced density, hence contributing less force to the cloth. The reduced density also allows for folds of greater size, as the cloth is less prone to the effects of gravity, and as such can fold more easily.

Part 5: Shaders!

Now that we have a simulation, we can now make it look even better by using something called shaders.

A shader program is essentially a set of instructions that tells a computer's graphics processing unit (GPU) how to render graphics. Shaders are small programs written in specialized shading languages that run directly on the GPU.

Vertex shaders and fragment shaders are two types of shaders that work together to create lighting and material effects in computer graphics.

1. **Vertex Shader:** This shader operates on each vertex (or point) of a 3D model. Its main job is to transform the 3D coordinates of each vertex from object space to screen space, which involves tasks like applying transformations such as scaling, rotation, and translation. Additionally, vertex shaders can perform other calculations like calculating lighting at each vertex or applying deformation effects. After processing each vertex, the vertex shader outputs the transformed vertex positions to the next stage of the rendering pipeline.
2. **Fragment Shader:** Also known as a pixel shader, the fragment shader operates on each pixel of the rasterized primitives (triangles, typically) generated by the vertex shader. Its main job is to calculate the final color of each pixel, taking into account factors such as lighting, texture mapping, and material properties. Fragment shaders can calculate lighting effects based on various lighting models, apply textures to surfaces, and perform other tasks like fog

or transparency effects. The output of the fragment shader is the final color of each pixel, which is then displayed on the screen.

Together, vertex and fragment shaders work in tandem to create realistic lighting and material effects in computer graphics. The vertex shader sets up the geometry and performs calculations at the vertex level, while the fragment shader handles pixel-level computations to determine the final appearance of each pixel on the screen. By combining these two types of shaders, developers can achieve a wide range of visual effects in real-time rendering applications.

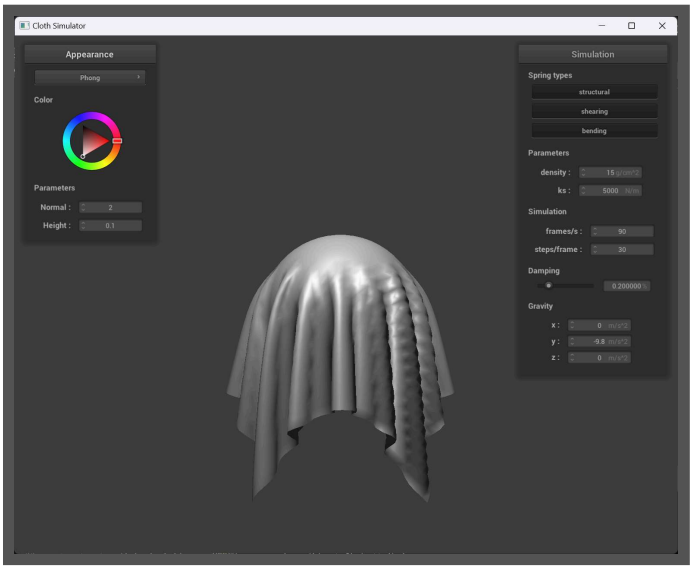
Blinn-Phong Shading

Blinn-Phong shading is a popular lighting model used in computer graphics to simulate the way light interacts with surfaces. It is an extension of the Phong reflection model. The Blinn-Phong model is widely used in real-time rendering applications due to its simplicity and performance characteristics.

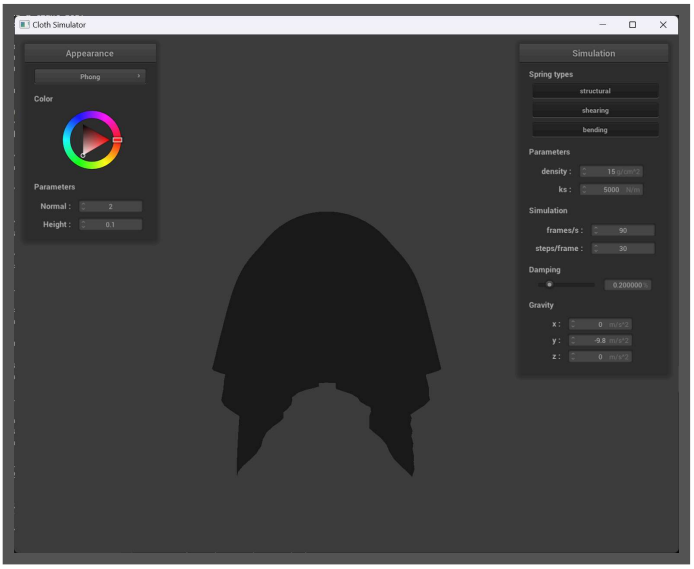
The Blinn-Phong shading model consists of three main components:

1. **Ambient Reflection:** This component represents the ambient light that is present in the scene. Ambient light is the light that is scattered and reflected by the environment, providing a base level of illumination for all objects in the scene. The ambient reflection term is typically a constant color value that is added to the final color of the object.
2. **Diffuse Reflection:** This component represents the light that is scattered uniformly in all directions by a surface. Diffuse reflection is responsible for the color and brightness of an object under direct lighting conditions. The diffuse reflection term is calculated based on the angle between the surface normal and the direction of the incoming light, as well as the intensity and color of the light source.
3. **Specular Reflection:** This component represents the light that is reflected in a specific direction by a surface. Specular reflection is responsible for highlights and shiny spots on an object's surface. The specular reflection term is calculated based on the angle between the direction of the reflected light and the direction of the viewer, as well as the shininess of the surface material.

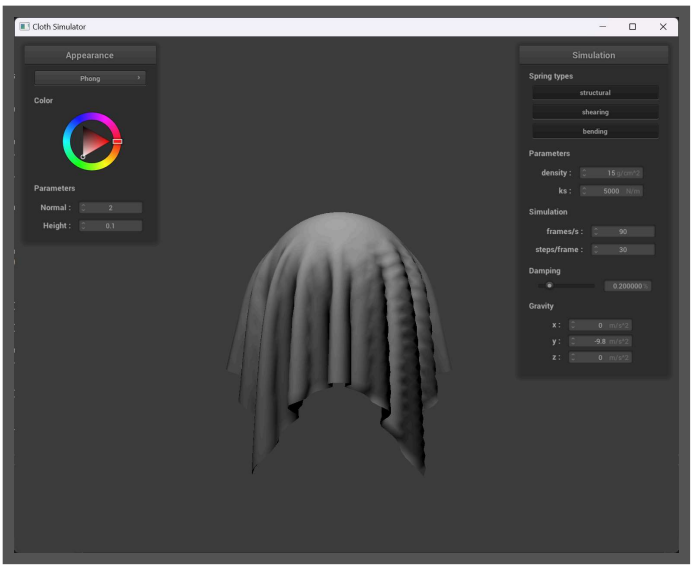
The Blinn-Phong shading model combines these three components to calculate the final color of a pixel on a surface. By adjusting the ambient, diffuse, and specular reflection coefficients, one can create a wide range of lighting effects, from matte surfaces to highly reflective materials.



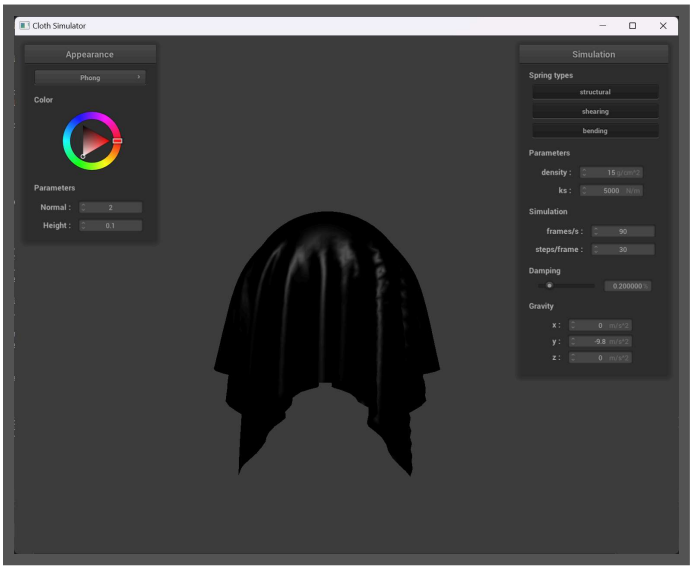
The full Blinn-Phong model



The ambient portion of the Blinn-Phong model



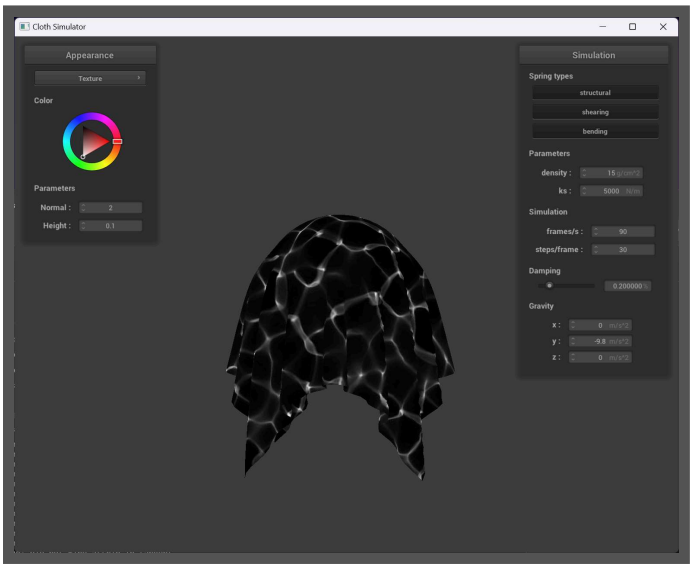
The diffuse portion of the Blinn-Phong model



The specular portion of the Blinn-Phong model

Texture Mapping

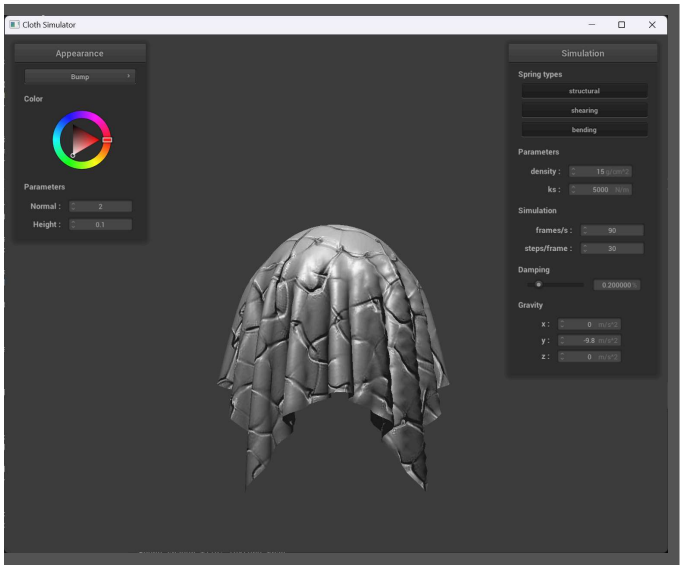
A texture mapping shader is a type of shader that is used to apply textures to a model. By calculating the corresponding texture coordinates for a given pixel, the shader can sample the texture and apply it to the model's surface.



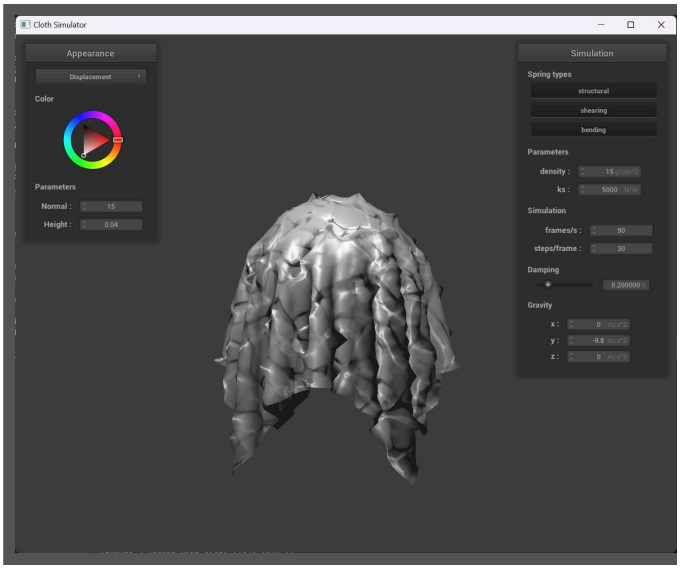
A texture of water caustics mapped onto the model of a sphere and a cloth

Bump and Displacement Mapping

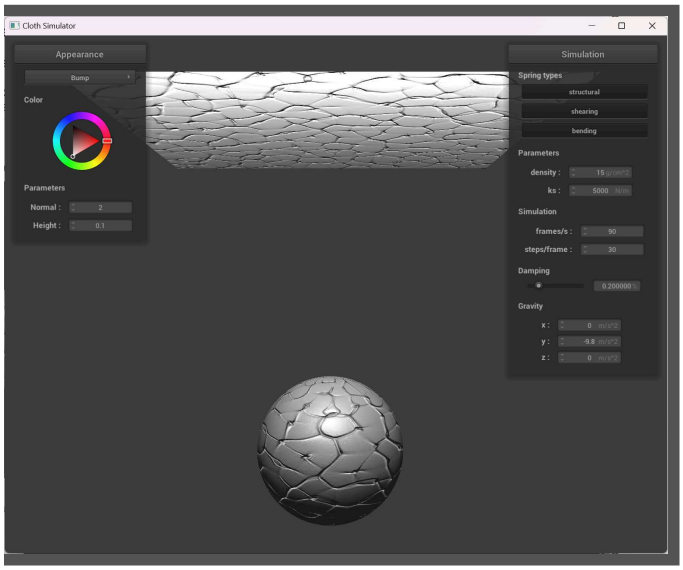
Bump (aka Normal) mapping utilizes Blinn-Phong shading combined with preprocessing step that recalculates the normals of the object based on a texture. This allows for the appearance of more detail on the object without actually changing the geometry of the object. Displacement mapping is similar to bump mapping, but it actually changes the geometry of the object based on the texture as well as opposed to just the normals. This allows for more realistic looking objects when the object is viewed in a manner such that the mapping would visibly change the geometry of the object.



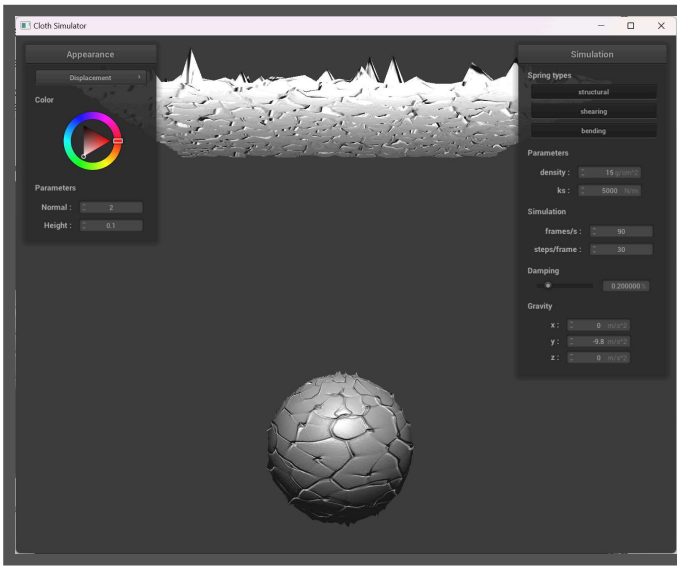
Bump Mapping on the cloth-covered sphere



Displacement Mapping on the cloth-covered sphere

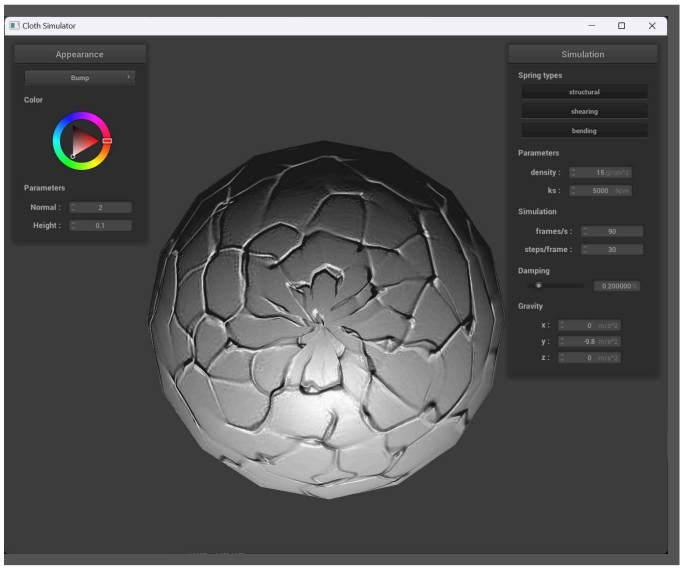


Bump Mapping

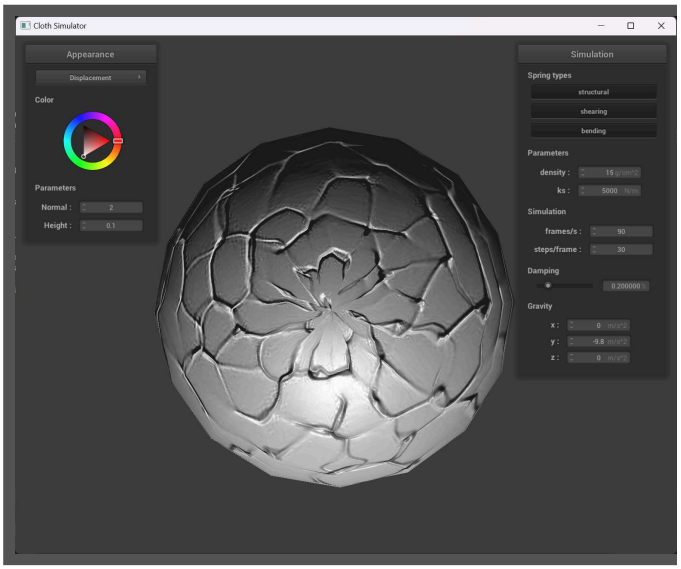


Displacement Mapping

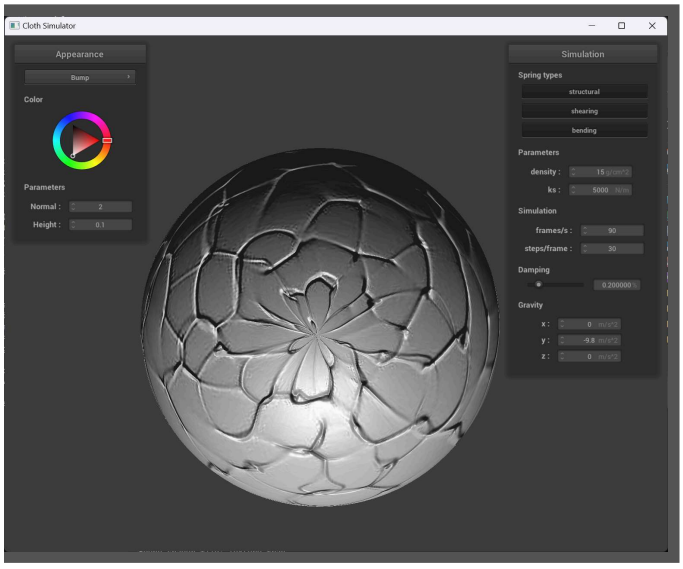
We can see a clear difference between the two images. The bump mapping image retains the same geometry as the original object, but the normals have been changed to give the appearance of more detail. The displacement mapping image, on the other hand, has actually changed the geometry of the object based on the texture. Do note that displacement mapping allows for a significant tuning, so the displacement can be much more pronounced or less pronounced than the image shown, with greater normal changes or less normal changes. We can see some interesting effects when we modify the mesh resolution of the sphere.



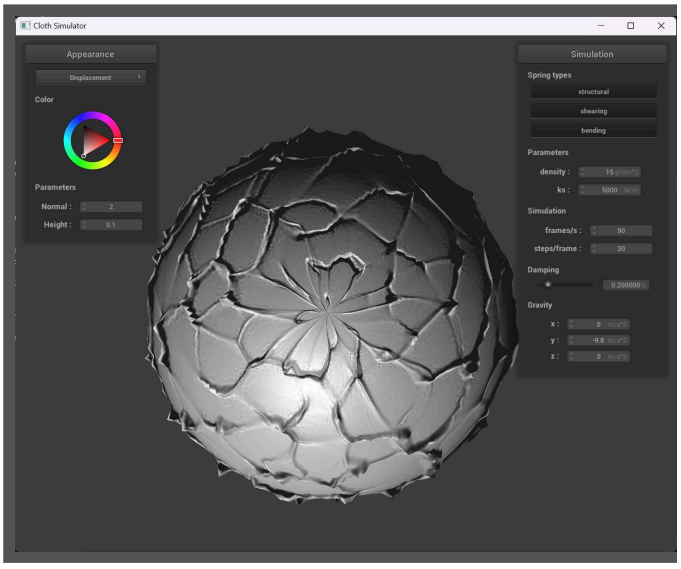
Bump Mapping with a Sphere of resolution 16



Displacement Mapping with a Sphere of resolution 16



Bump Mapping with a Sphere of resolution 128



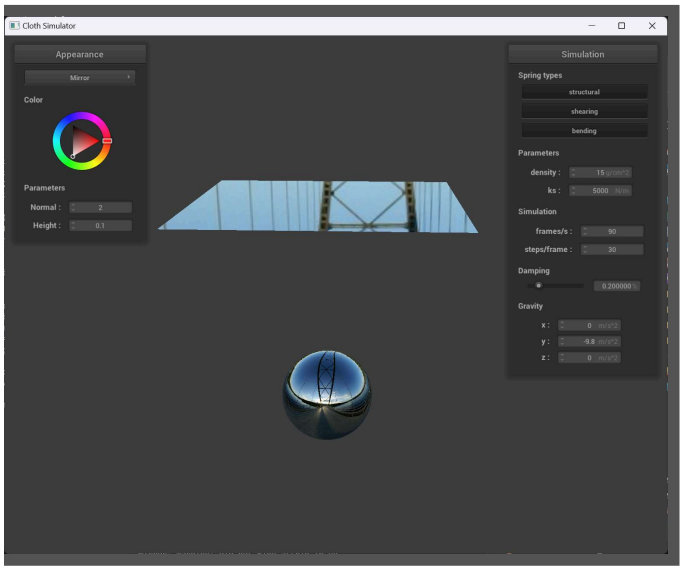
Displacement Mapping with a Sphere of resolution 128

We can see that while the bump mapping at both sphere resolutions look as one would expect (the sphere is simply smoother), the displacement mapping gains more detail as the resolution of the sphere increases. We can see that the displacement mapping at a resolution of 128 has a much

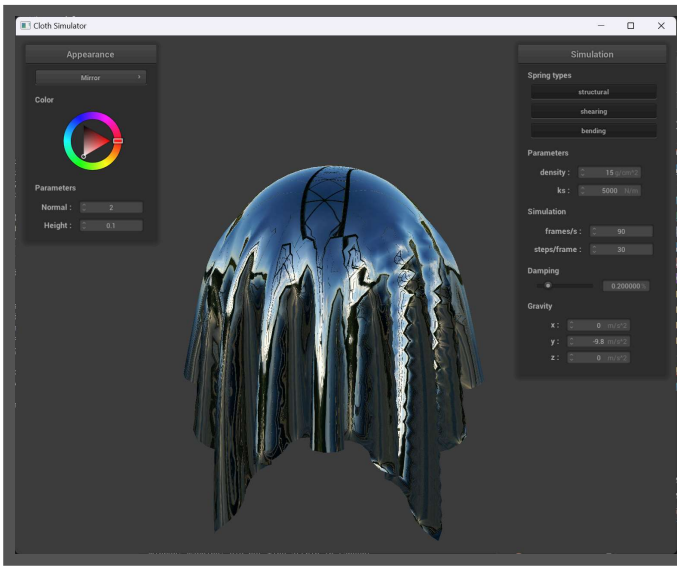
more pronounced effect than the displacement mapping at a resolution of 16. This is because the displacement mapping actually changes the geometry of the object, so the more geometry there is to change, the more pronounced the effect will be!

Mirror Shading

A mirror shader can be created by computing the ray in and ray out vectors and sampling something called a cube map. A cube map is a 6 sided texture that is used to simulate reflections, effectively providing a source of bounced light.



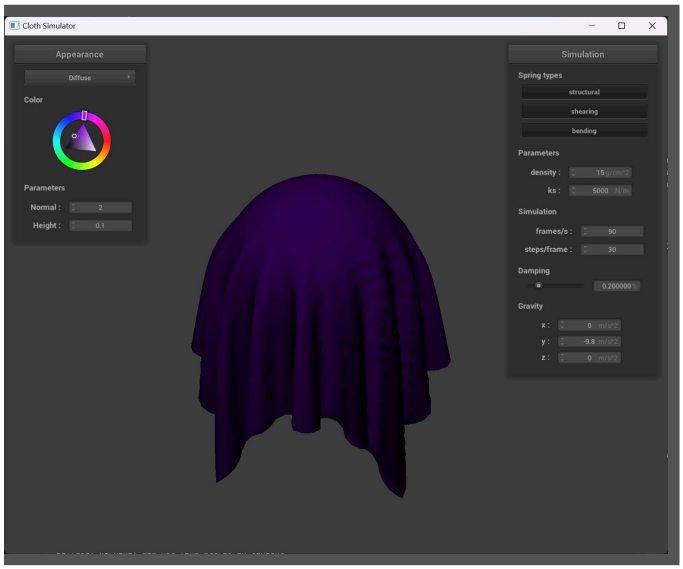
The mirror shading applied to a sphere and a plane



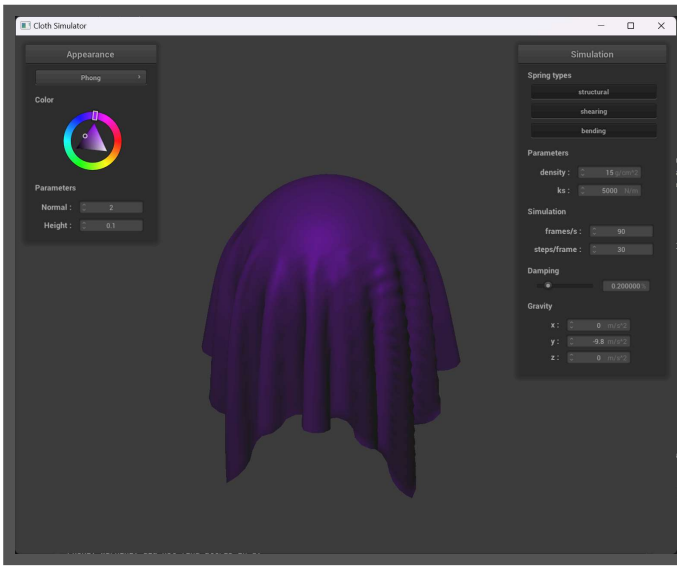
Mirror shading applied after the cloth has fallen

Extra Credit: Adding Color!

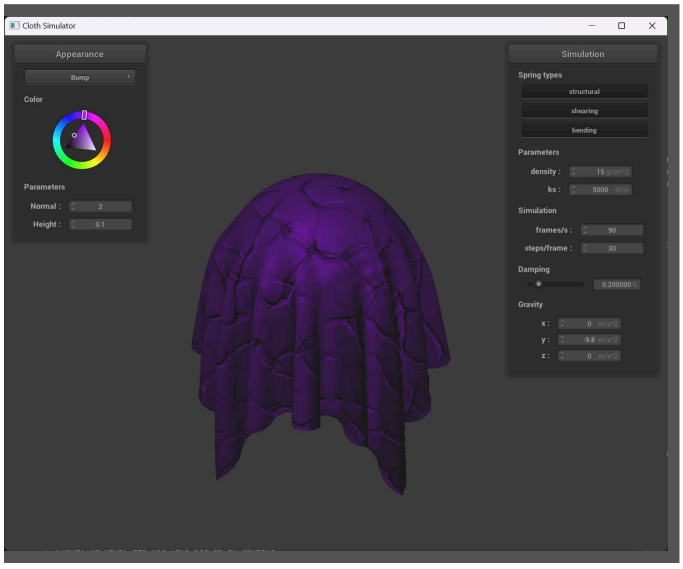
We can add texture to our previous shaders to make them look even better! This can be done by modifying our diffuse component of our diffuse, phone, bump, and displacement shaders to be multiplied by the input color. Nothings better than a cloth that's purple!



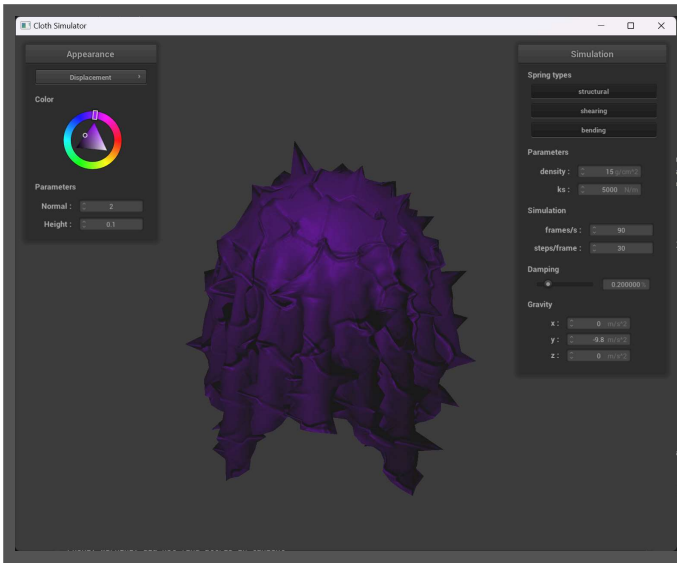
A purple, diffuse cloth!



A purple, Phong-shaded cloth!



A purple, bump-mapped cloth!



A purple, displacement-mapped cloth!