CS 184: Computer Graphics and Imaging, Spring 2023

Project 4:

Brian Santoso, CS184 Team: #1-lana-del-rey-fan!!!

Webpage: https://briansantoso.github.io/project-webpages-sp23-BrianSantoso/proj4/index.html

Overview

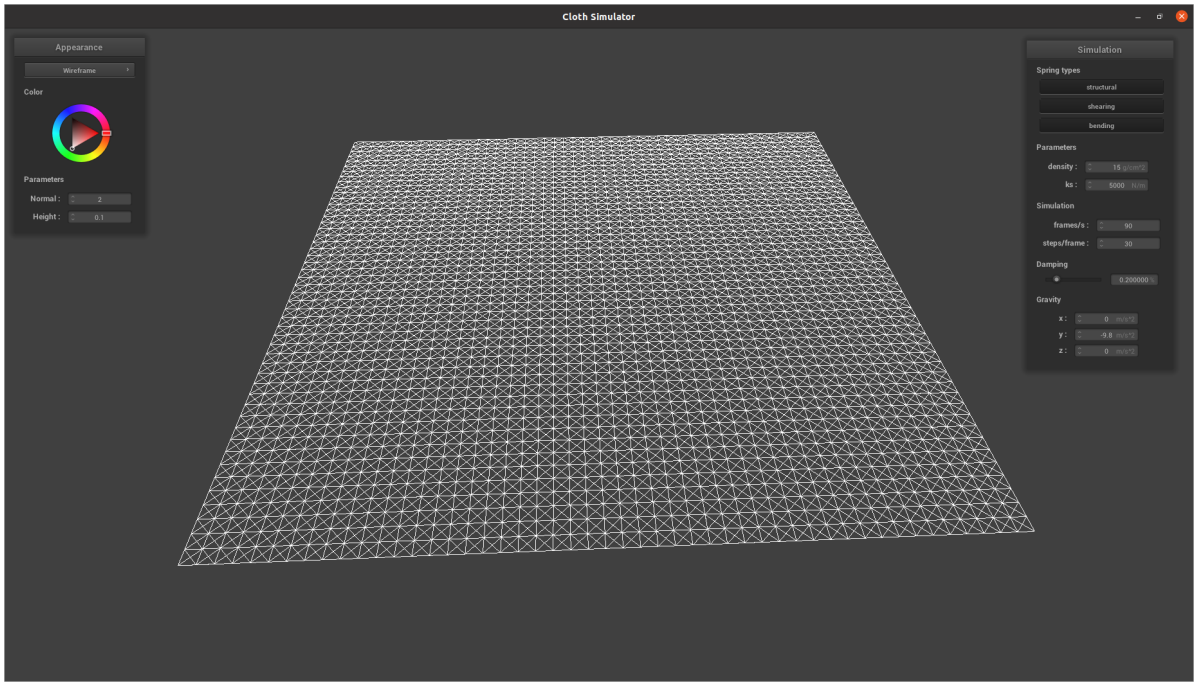
In this project, I implement a cloth simulator with object collisions and self-collisions. The cloth is comprised of a grid of point masses with structural, shearing, and bending springs. I simulate the movement of cloth with external forces (i.e., gravity) and internal spring forces via Verlet integration. I handle cloth collisions with spheres and planes. Finally, I implement various vertex and fragment shaders, including Blinn-Phong shading, texture mapping, bump mapping, displacement mapping, and mirror material with environment-mapped reflection shaders.

Part 1

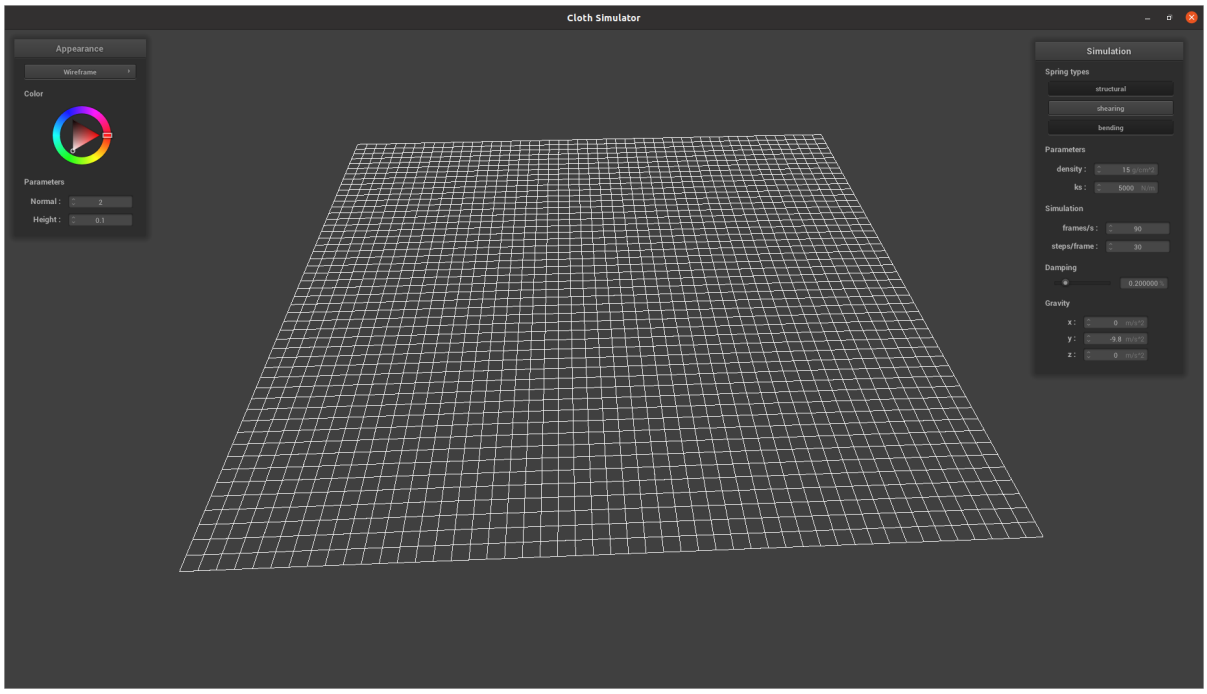
In this part, I build a grid of springs and masses. First I build the grid of masses by simply iterating over the width and height of the cloth and adding masses in a grid with `length / (num_points - 1)` space between adjacent point masses with respect to width and length.

Then I build the springs connecting the masses. For each point mass in the grid, whenever possible, I construct springs connecting the mass to its adjacent 1 to the right, 1 above (structural springs), 2 to the right, 2 above (bending springs), diagonal up and right 1, and finally a spring connecting the 2 masses directly above and to the right of the current mass (shearing springs).

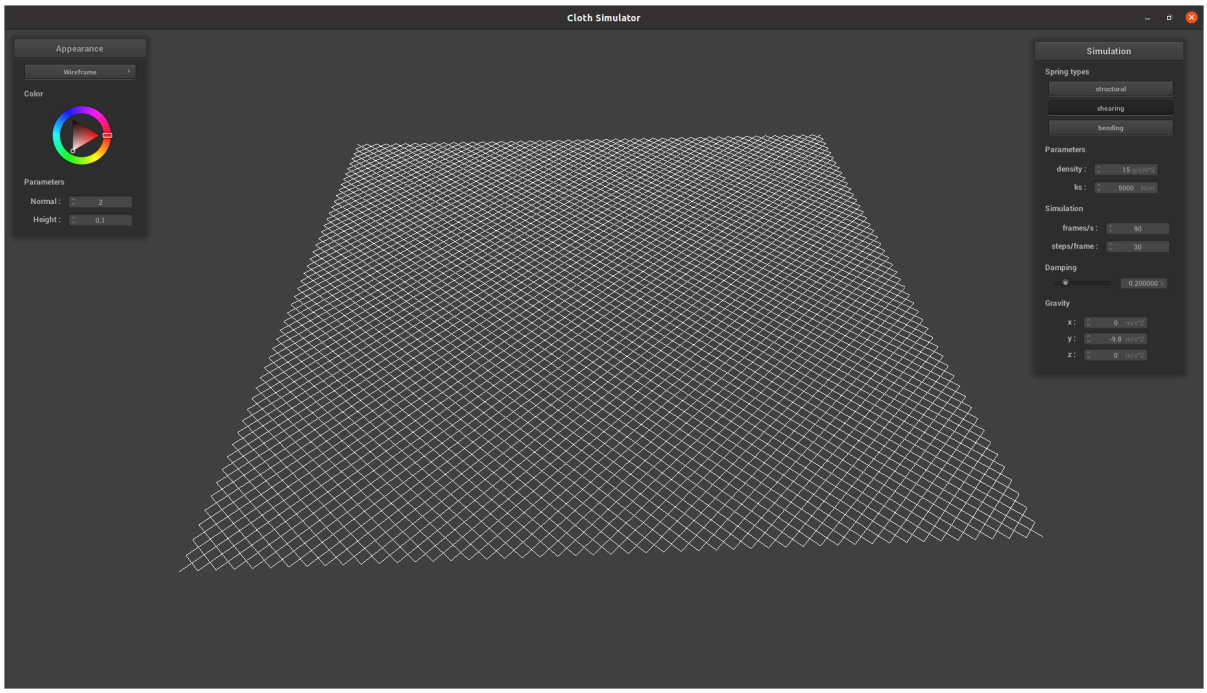
- **Take some screenshots of *scene/pinned2.json* from a viewing angle where you can clearly see the cloth wireframe to show the structure of your point masses and springs.**



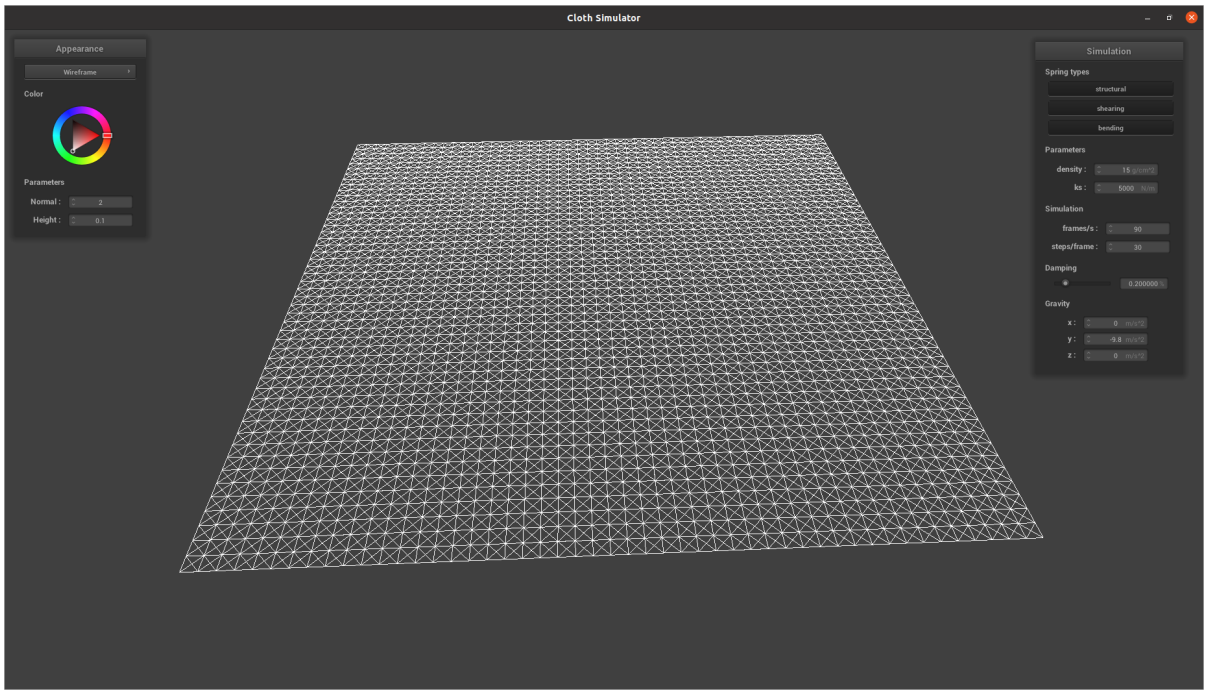
- **Show us what the wireframe looks like (1) without any shearing constraints, (2) with only shearing constraints, and (3) with all constraints.**



Without shearing



Only shearing



All constraints

Part 2

In this part I simulate the movement of cloth with external forces (i.e., gravity) and internal spring forces via Verlet integration, without object collisions and self-collisions (implemented in parts 3 and 4).

At each timestep, I reset the external forces of all masses as not to accumulate extra, non-existent forces.

Then, I compute the net external force to be applied to all point masses once since all external forces are applied uniformly to all masses—this is accomplished by adding up all the external accelerations and multiplying by mass. I then apply this force to all masses.

Then I calculate the net spring correction forces for each individual mass. I do this by iterating through all springs, and accumulating the resulting spring correction forces to each of the spring's 2 masses (the 2 resulting forces are equal and opposite each other). These spring correction forces are calculated by scaling the respective direction of the mass by the spring constant and the difference between the distance between the masses and spring rest length. For bending constraint springs, the spring constant is multiplied by 0.2.

Now that I have calculated the net force acting upon each mass, I then update the position of each mass via Verlet integration, ignoring pinned masses. Verlet integration is a method of numerical integration. It is an explicit integrator that is fairly accurate and relatively easy to implement. The updated position of each point mass is described as follows:

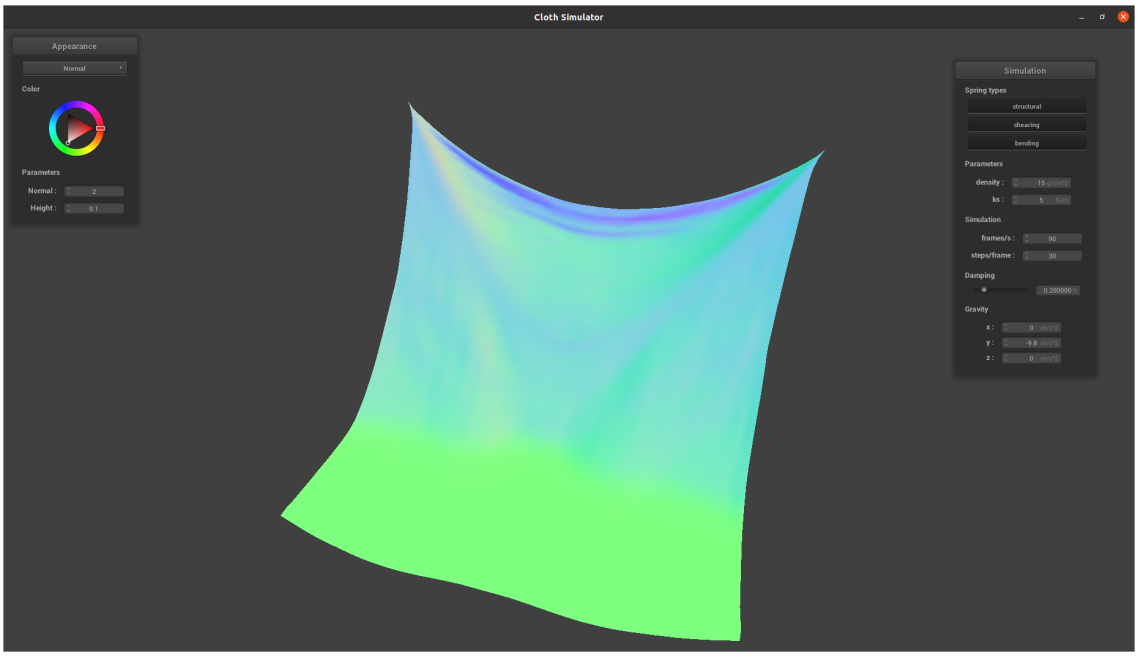
$$x_{t+dt} = x_t + (1 - d) * (x_t - x_{t-dt}) + a_t * dt^2$$

Where `d` is the dampening constant, `x_t - x_{t-dt}` is the difference in position from the current time step and previous timestep and is equal to `v_t * dt`, and `a_t` is the acceleration at the current timestep which is calculated by dividing the point mass's net force by its mass.

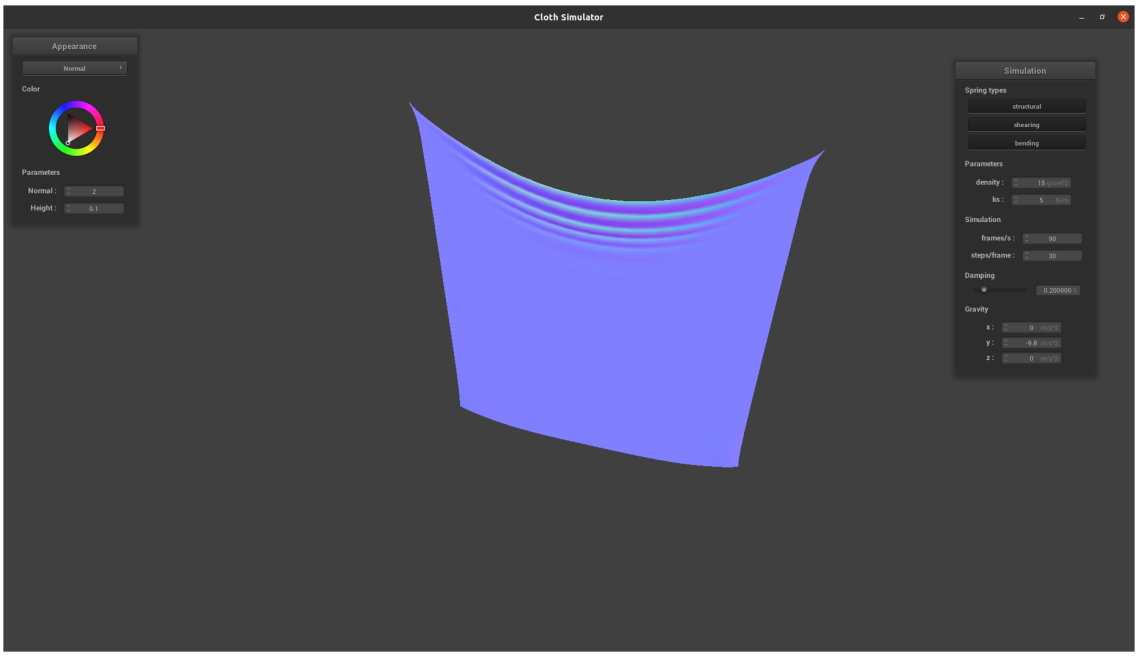
Finally I constrain the changes in the point mass positions to be such that the spring does not change in length more than 10% per timestep according the 1995 SIGGRAPH Paper [Provot 1995]. If the spring length is more than 10% of its rest length, then I apply a position correction uniformly to each of the non-pinned masses. The magnitude of the total position correction is equal to the difference in the spring's current length and max length (1.1 * rest length).

- **Experiment with some the parameters in the simulation. To do so, pause the simulation at the start with `P`, modify the values of interest, and then resume by pressing `P` again. You can also restart the simulation at any time from the cloth's starting position by pressing `R`.**
    - **Describe the effects of changing the spring constant `ks`; how does the cloth behave from start to rest with a very low `ks`? A high `ks`?**

      A small spring constant corresponds to a "springy" cloth, and before the cloth reaches its rest state, the cloth has many wrinkles and waves. At its rest state, the cloth has multiple folds between its pinned masses when it droops.



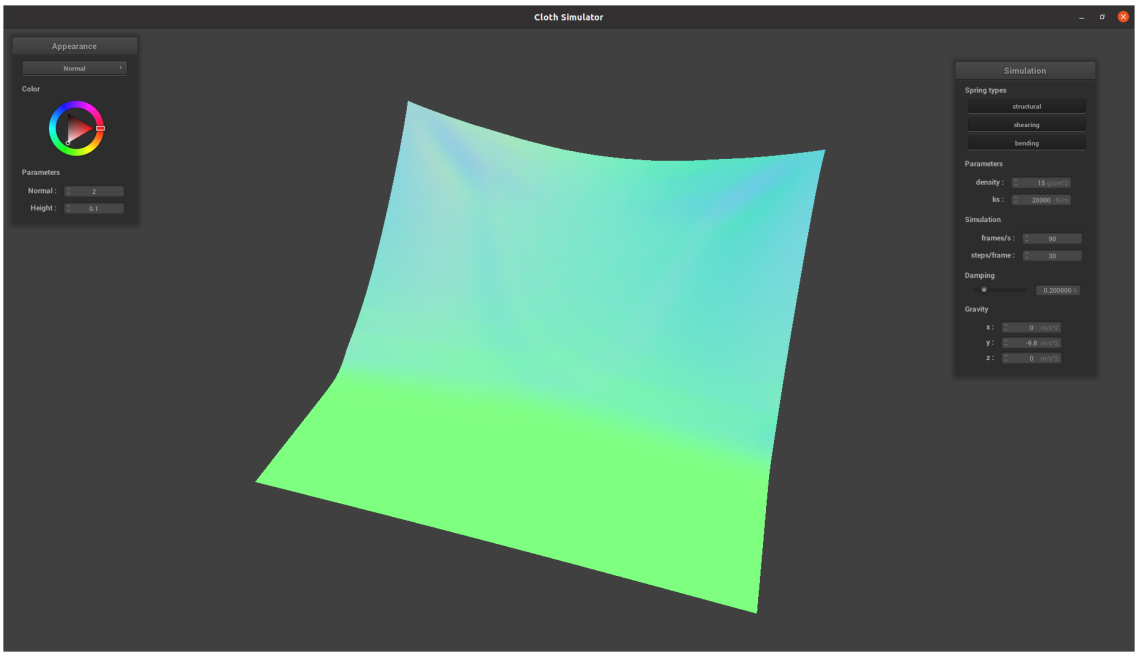
Small spring constant `ks = 5`



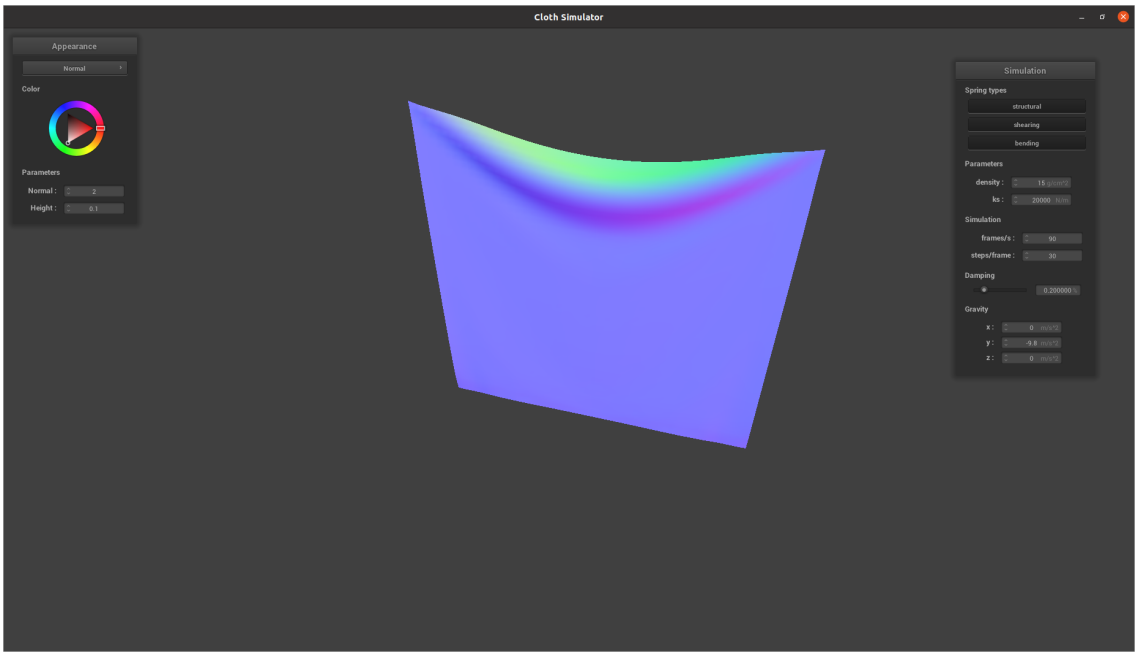
Small spring constant `ks = 5`

A large spring constant corresponds to a “stiff” cloth, and before the cloth reaches its reset state, the cloth has few wrinkles and waves, and appears

smoother in texture. At its rest sate, the cloth has few (often only a single) fold between its pinned masses when it droops.



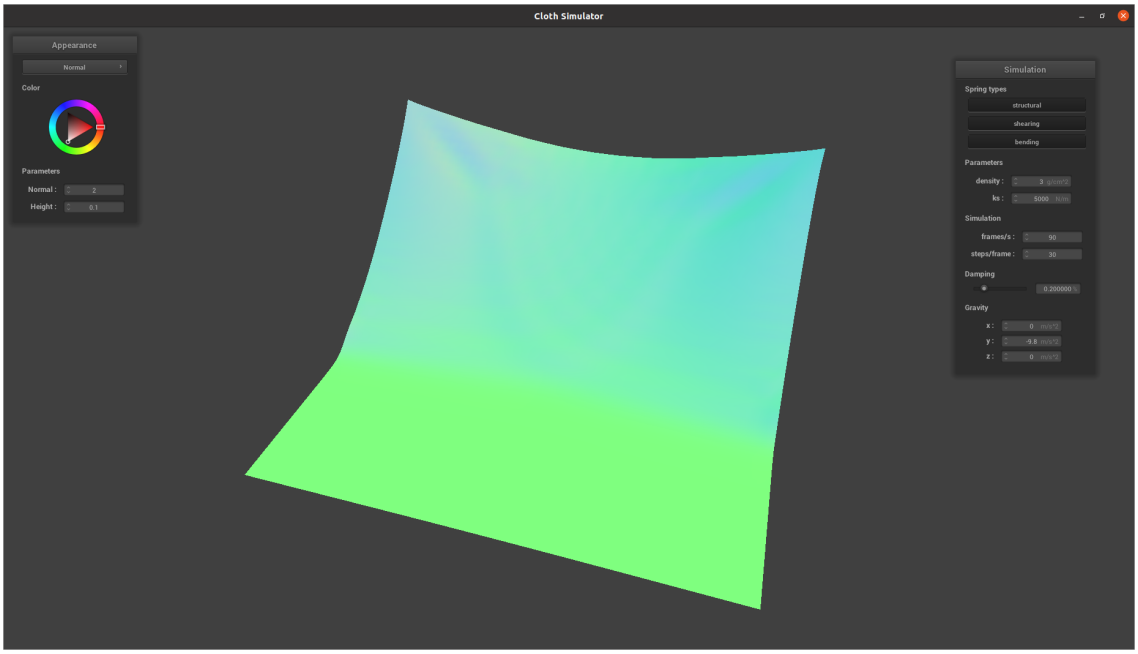
Large spring constant `ks = 20000`



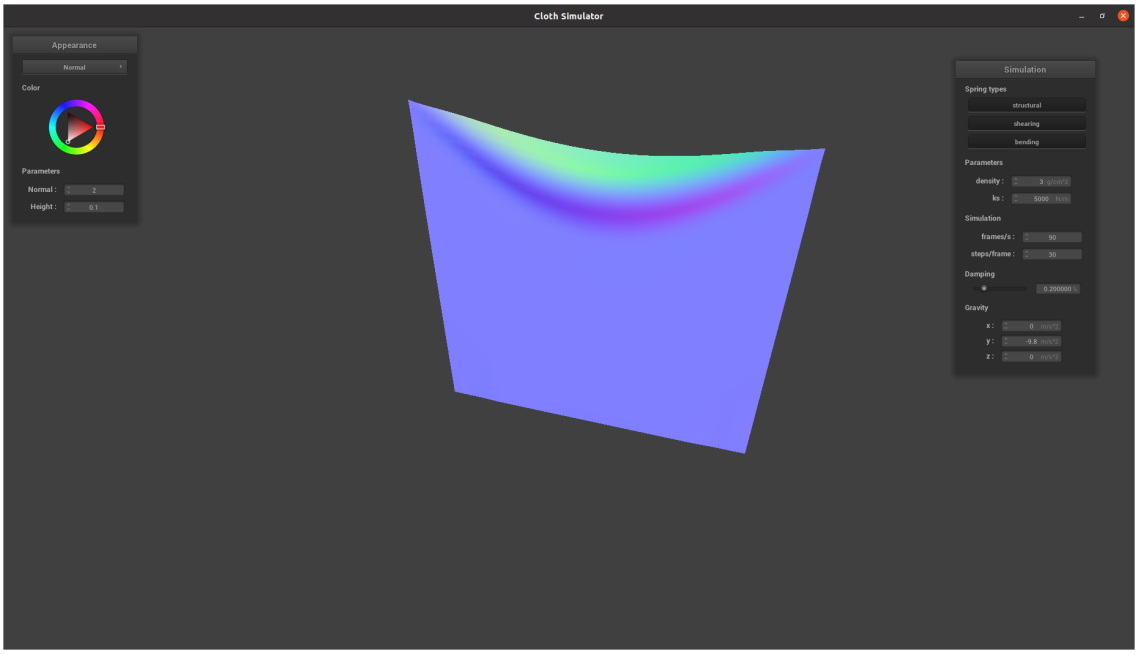
Large spring constant `ks = 20000`

- **What about for `density`?**

A small density corresponds to a “light” cloth, and before it reaches its rest state, the cloth has few wrinkles and waves - it appears as if it falls smoothly. At its rest state, the cloth droops down less between its pinned masses due to its lighter weight and has few wrinkles.

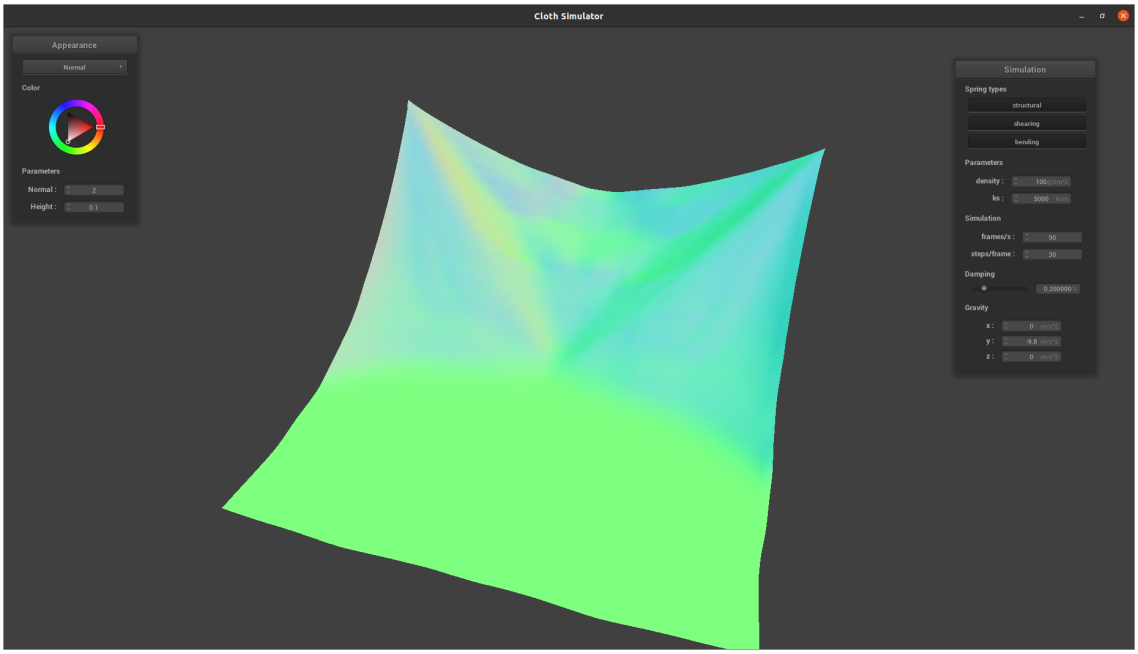


Small density

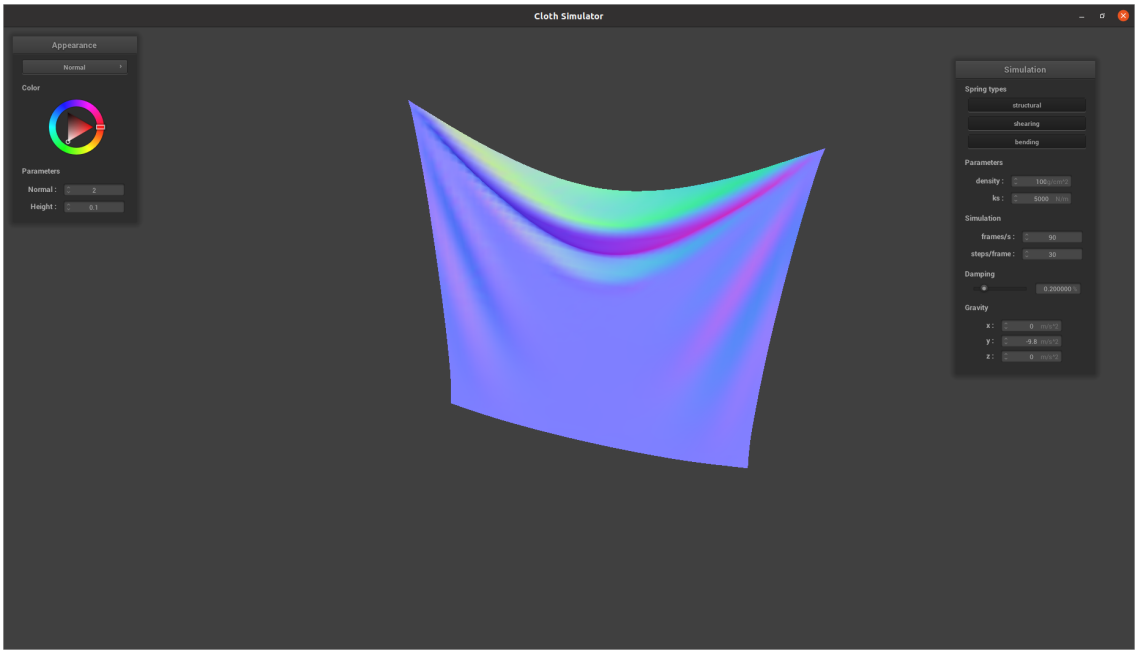


Small density

A high density corresponds to a “heavy” cloth, and before it reaches its rest state, the cloth has many wrinkles and waves - it appears as if it falls violently. At its rest state, the cloth droops down more between its pinned masses due to its heavier weight and has many wrinkles.



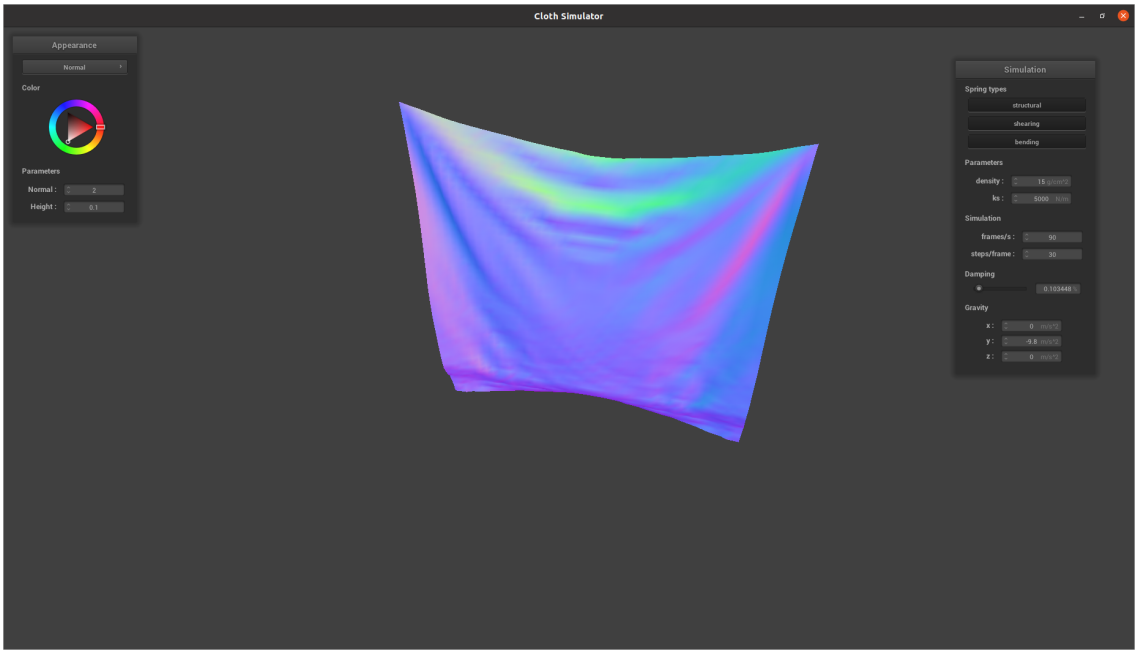
High Density



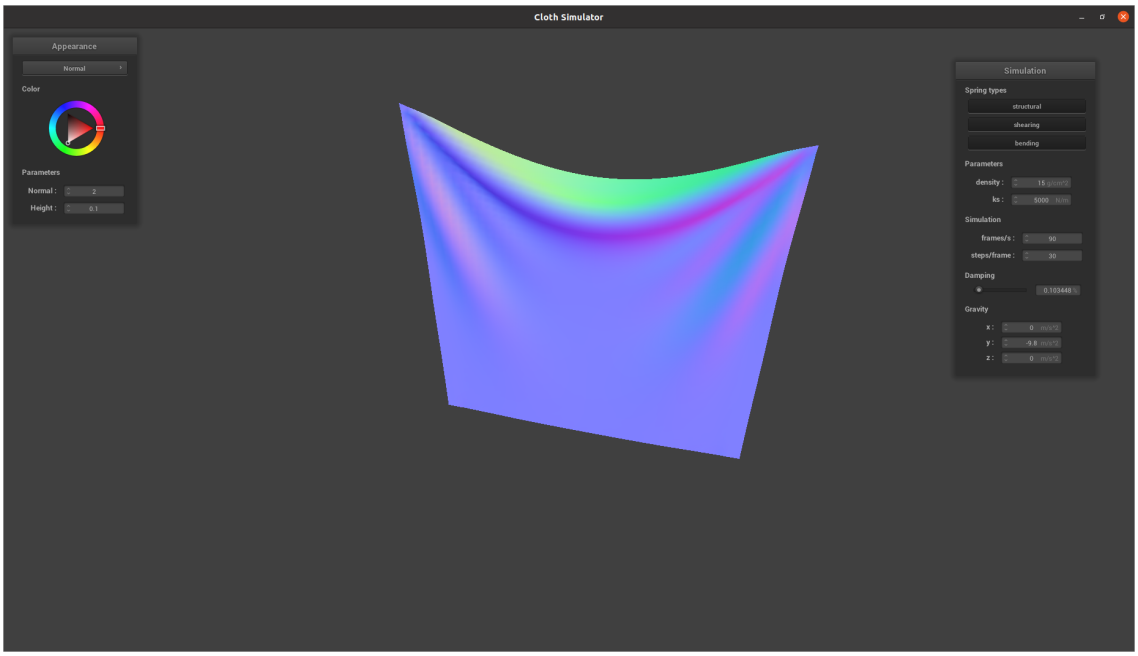
High Density

- **What about for `damping`?**

  A small dampening value corresponds to a “stretchy”/”bouncy” cloth, and before it reaches its rest state, the cloth falls quickly, chaotically, manifesting many wrinkles and waves - it bounces around for a long time before finally reaching its rest state and almost appears fluid-like. At its rest state, the cloth has few folds between its pinned masses.

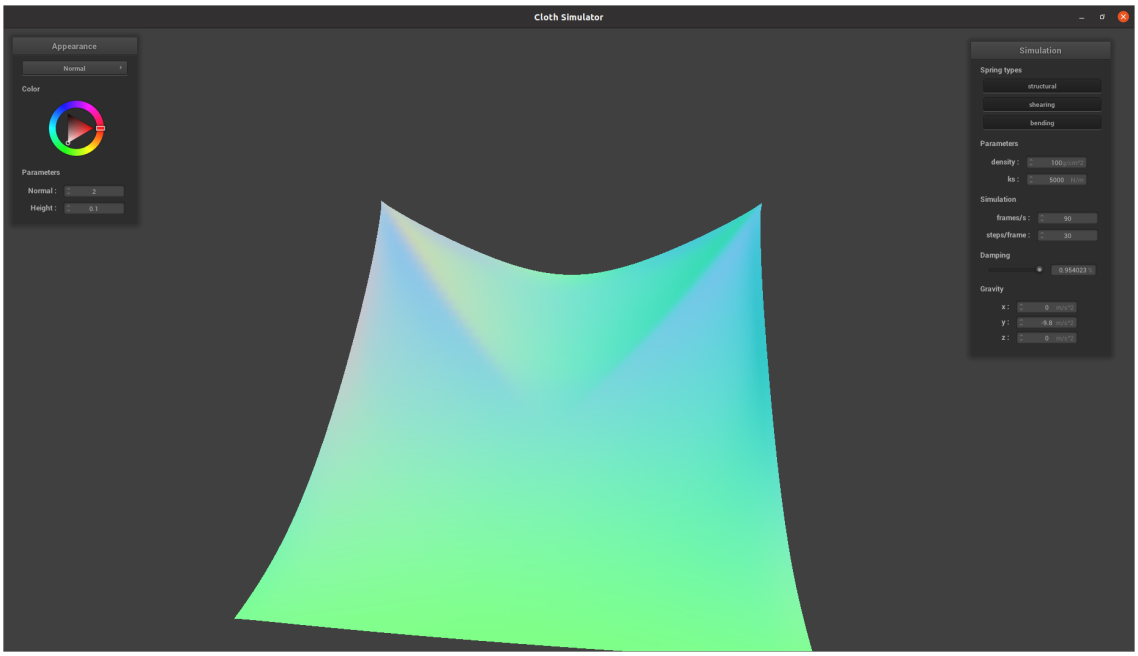


Small damping

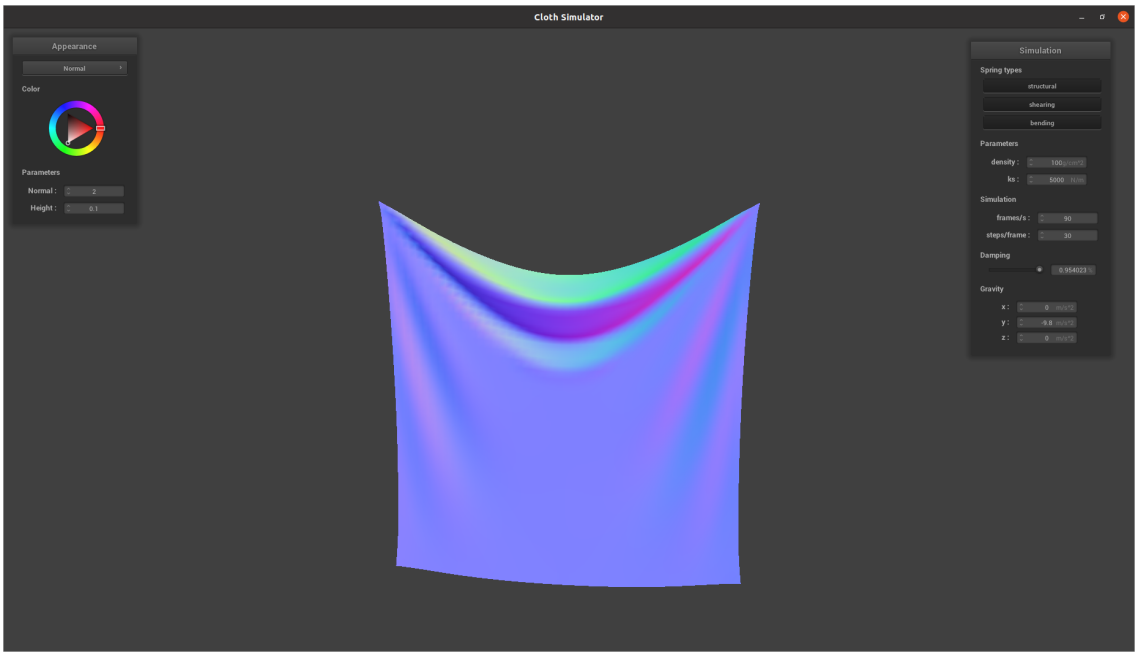


Small damping

A large dampening value corresponds to a “stiff”/”firm” cloth, and before it reaches its rest state, the cloth falls slowly, smoothly, and with few wrinkles or waves - it does not bounce for long before reaching its rest state and appears more solid-like. At its rest state, the cloth has few folds between its pinned masses.

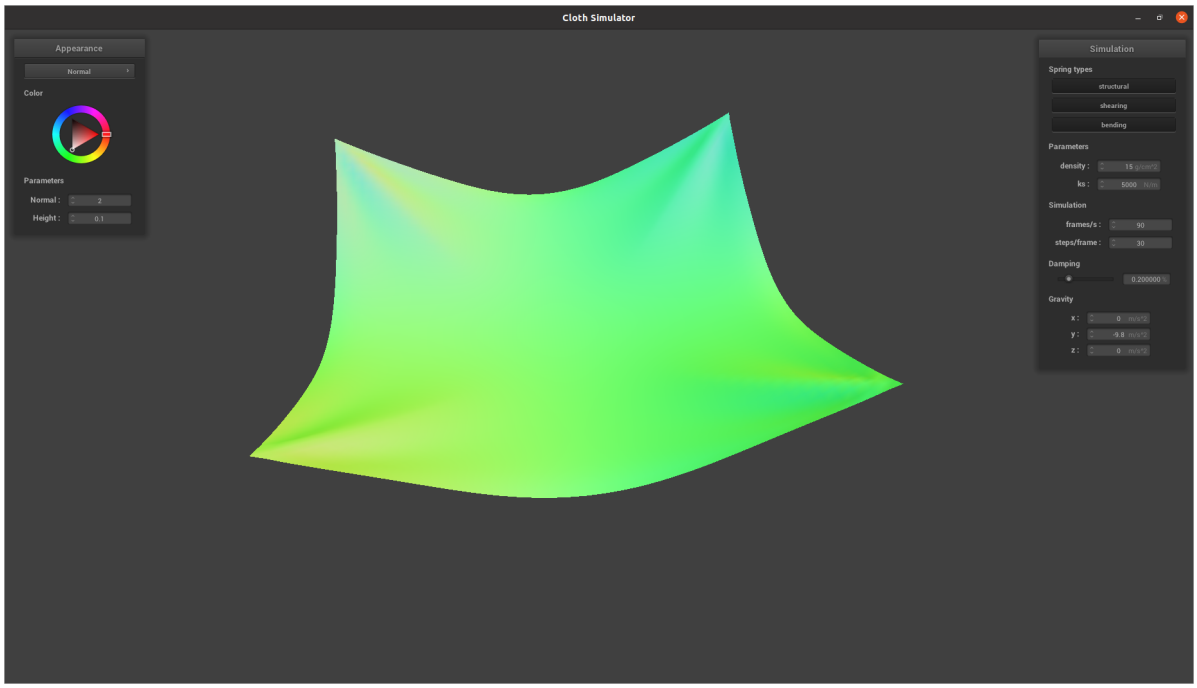


Large damping



Large damping

- **Show us a screenshot of your shaded cloth from *scene/pinned4.json* in its final resting state! If you choose to use different parameters than the default ones, please list them.**



Part 3

In this part, I handle collisions with other objects, namely spheres and planes.

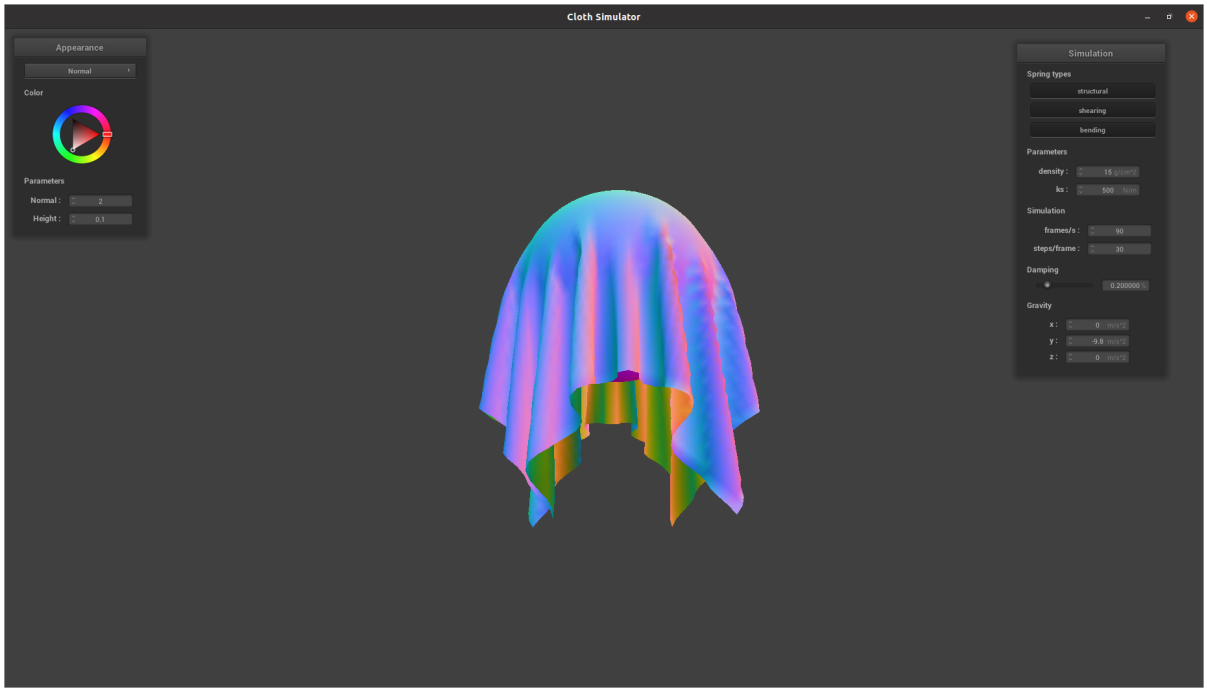
For collisions for spheres, I first check if the mass is inside the sphere by simply checking if its distance to the sphere's origin is less than or equal the radius. If the mass is inside the sphere, then I compute the tangent point which is defined as where the point mass should have intersected the sphere. The direction of this path is calculated as the difference between the mass position and the sphere's origin. Then I compute the correction vector that needs to be applied to the mass's last position in order to reach the tangent point, scaled down by friction (1 - f).

Show us screenshots of your shaded cloth from *scene/sphere.json* in its final resting state on the sphere using the default `ks = 5000` as well as with `ks = 500` and `ks = 50000`. Describe the differences in the results.

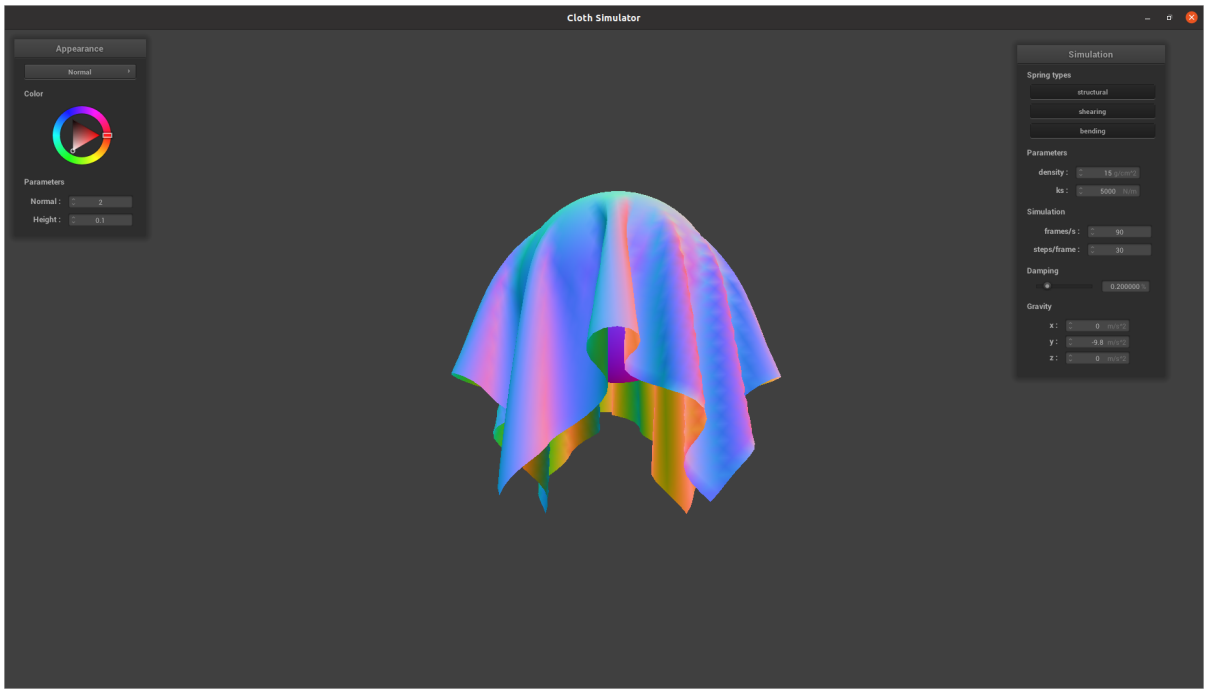
With `ks = 500` the cloth droops the most with many folds and waves, with its folds appearing the most intensely creased.

With `ks = 5000` the cloth droops wider with a less folds and waves, appearing less intensely creased.

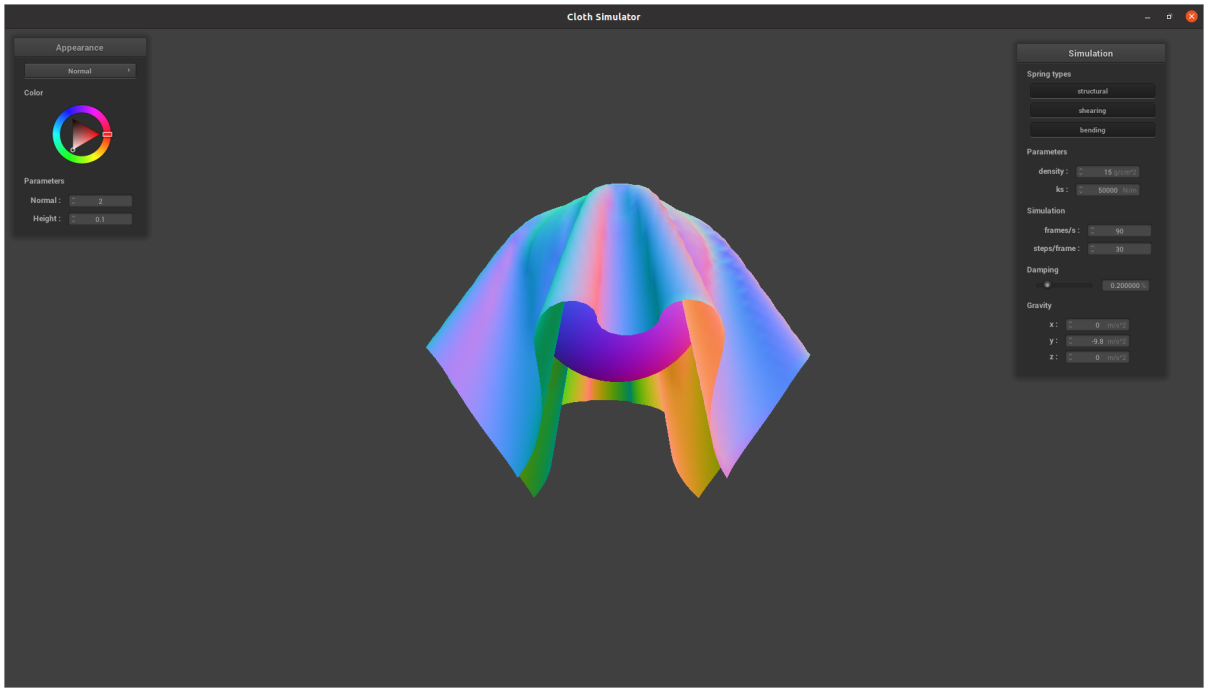
With `ks = 50000` the cloth droops the widest with the fewest folds and waves, appearing the least intensely creased.



`ks = 500`

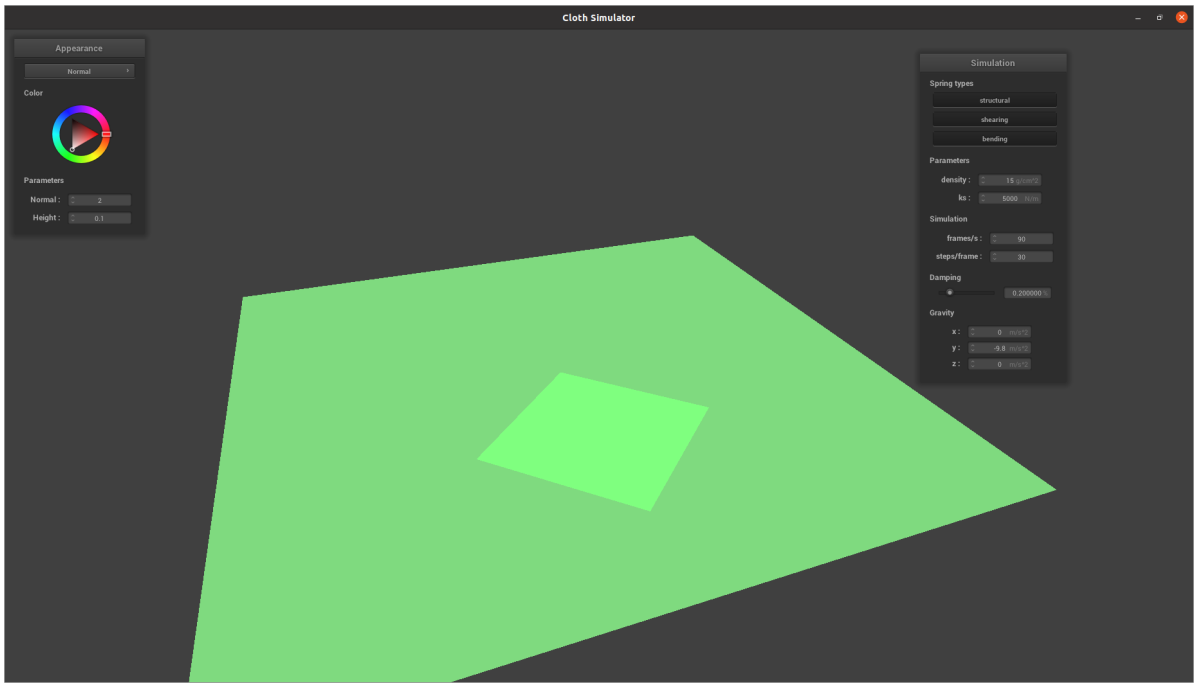


`ks = 5000`



`ks = 50000`

- **Show us a screenshot of your shaded cloth lying peacefully at rest on the plane.**



Part 4

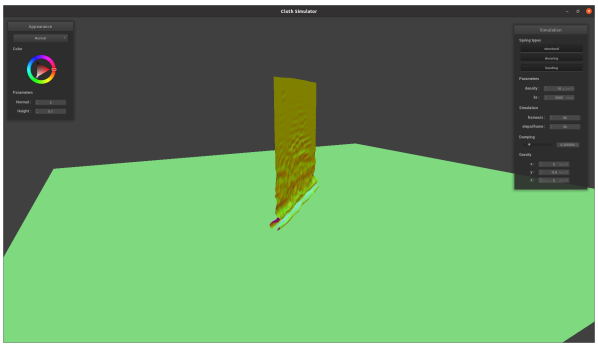
In this part I handle self-collisions of the cloth using a spatial hash map to accelerate comparisons. First I define a hash function maps a 3D position into a unique float identifier that represents membership in a 3D box volume. To do this, I first define a 3D box volume with width, height and depth `w`, `h`, and `t`. The hash function takes a 3D position and computes “box coordinates” (x,y,z indices of the corresponding a 3D box volume) by floor dividing by the box volume dimensions. The hash function then returns `box_x * 256 + box_y * 16 + box_z` to achieve a float with few hash collisions.

Then I construct a spatial map that maps these unique float identifiers to a list of point masses belonging to the corresponding 3D box volume.

Finally, to implement the self-collision handling for a given point mass, I retrieve the list of masses that belong to the same box volume. For each mass in the volume, not including the original mass itself, a collision is defined as having a distance within twice the thickness of the cloth. If a collision occurs, then a correction vector is accumulated to the original mass such that the two colliding point masses are not colliding anymore. The final position correction vector for the original mass is equal to the net correction,

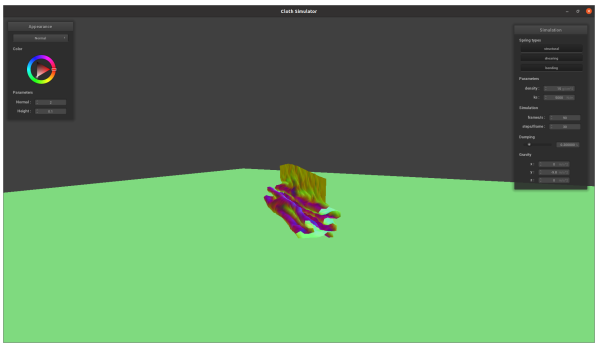
divided by the number of collisions, and scaled down by the number of simulation steps. Finally, this position correction is applied to the original point mass.

- **Show us at least 3 screenshots that document how your cloth falls and folds on itself, starting with an early, initial self-collision and ending with the cloth at a more restful state (even if it is still slightly bouncy on the ground).**



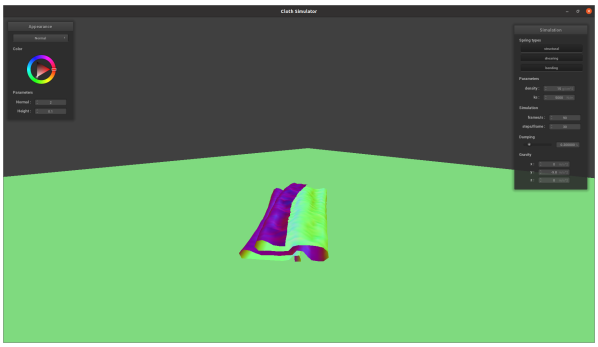
Falling cloth at its first self-collision

At its first self-collision, the cloth folds at the bottom and a wave of ripples propagates towards the top of the cloth.



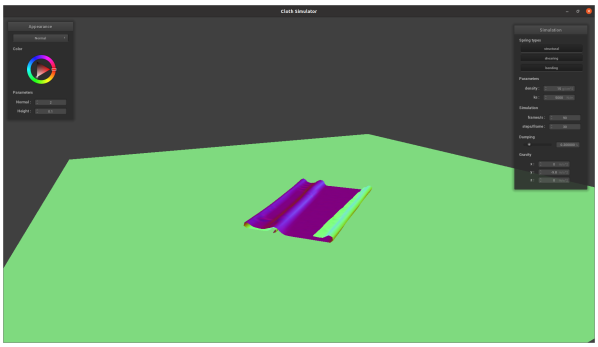
Falling cloth after many early self-collisions.

The cloth accumulates folds as the falls onto itself.



Falling cloth begins to flatten out.

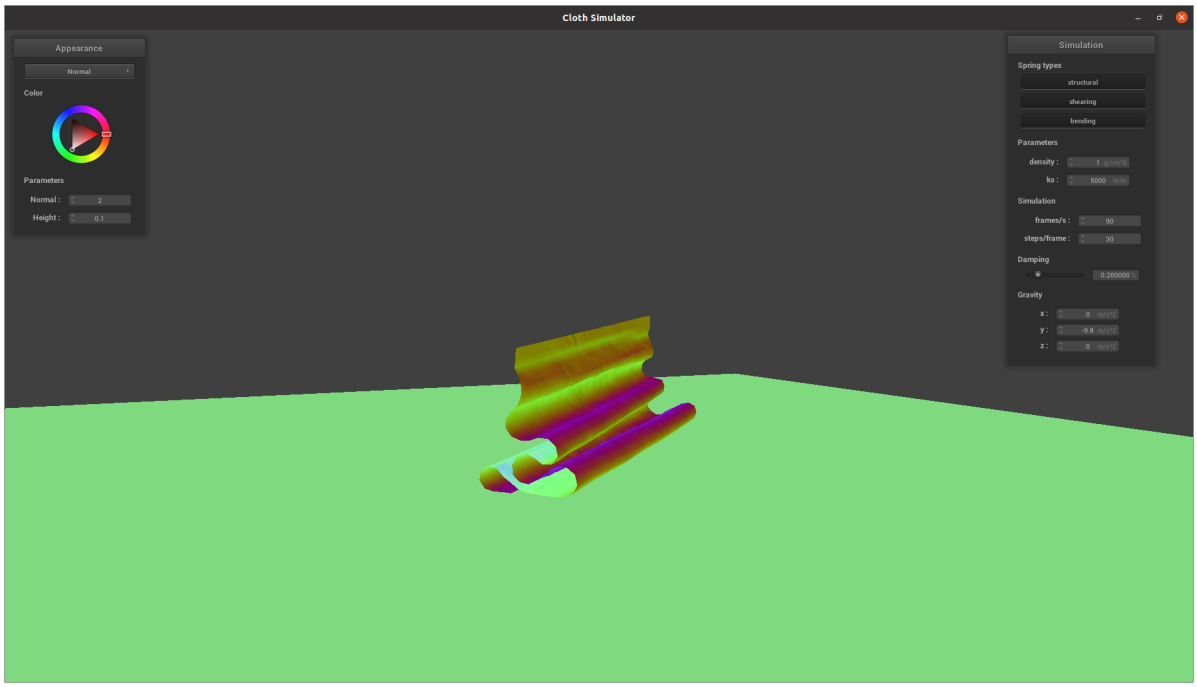
Ripples and wrinkles begin to smooth out.



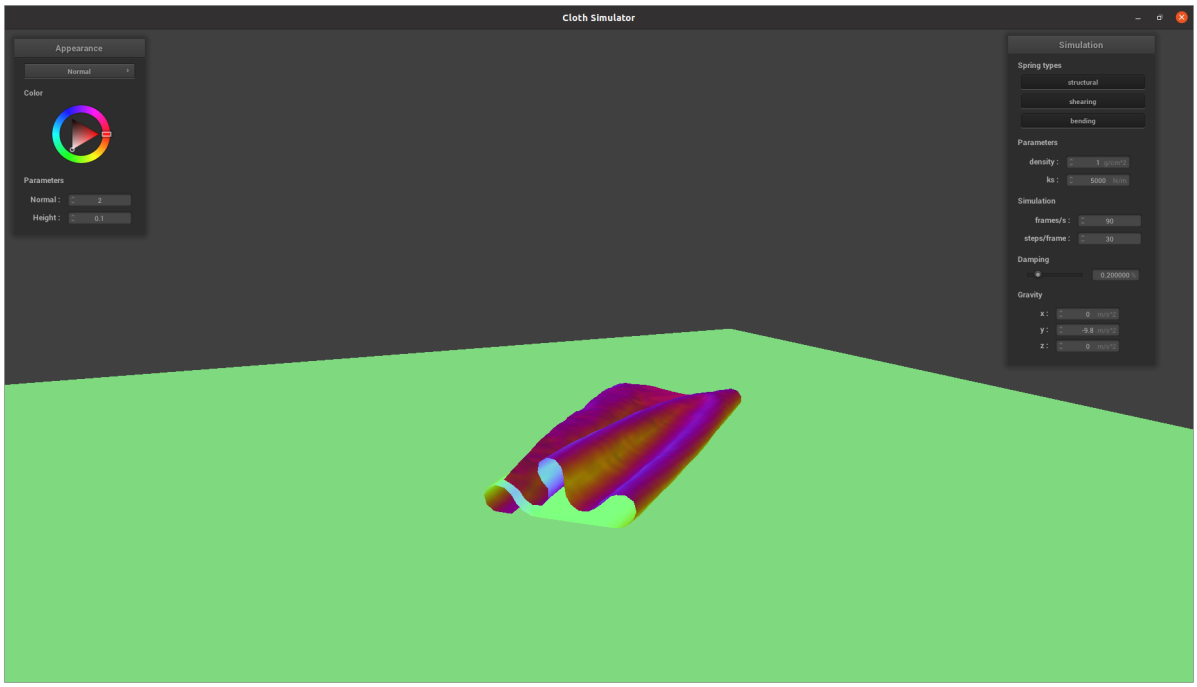
Falling cloth at its restful state. The cloth is smooth, resting peacefully on the ground (albeit with a little bouncing).

- **Vary the `density` as well as `ks` and describe with words and screenshots how they affect the behavior of the cloth as it falls on itself.**

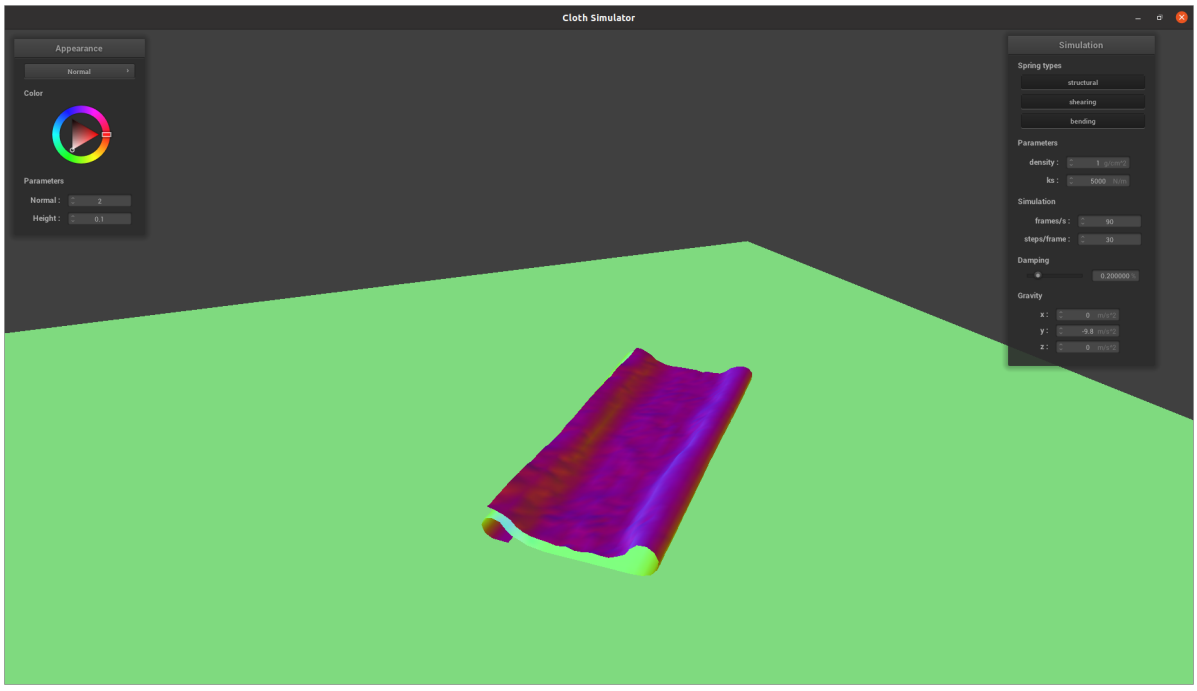
With a low density, many thin waves propagate quickly through the cloth after its first self-collision. The cloth falls slowly and bounces on itself when self-colliding. It continues to bounce quite a bit before becoming flat. When the cloth is flat, there are many thin fluid-like ripples that propagate throughout the cloth before it arrives at its resting state



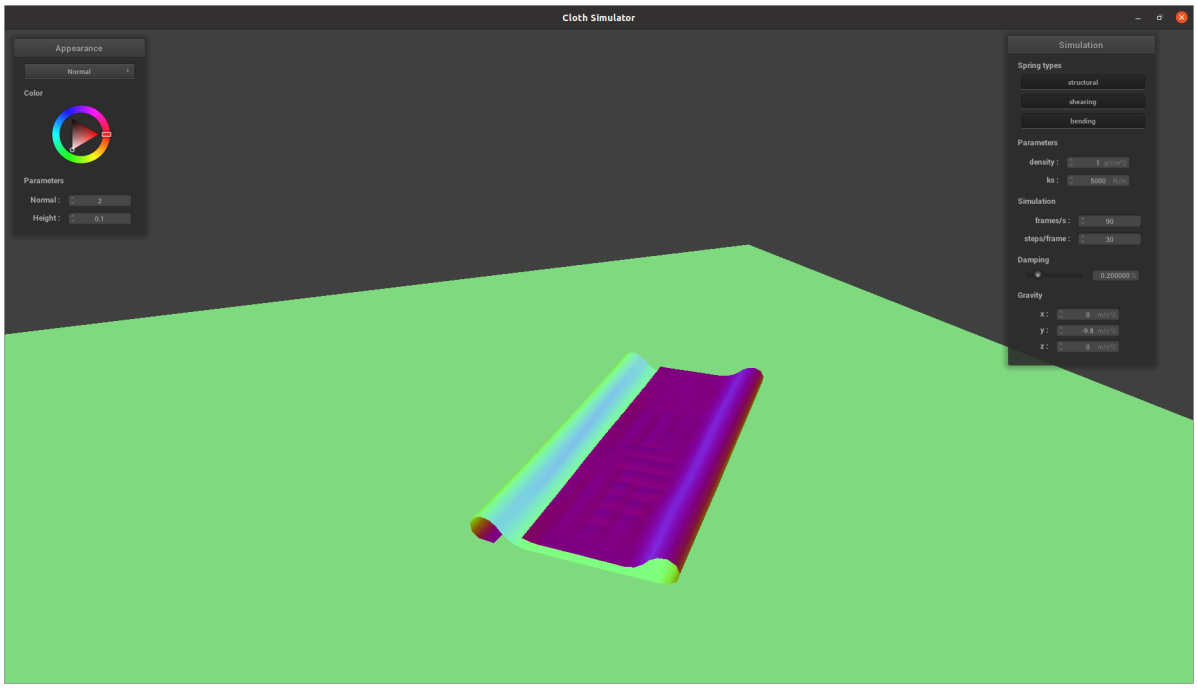
Low density cloth begins to self collide



Low density cloth falls to floor

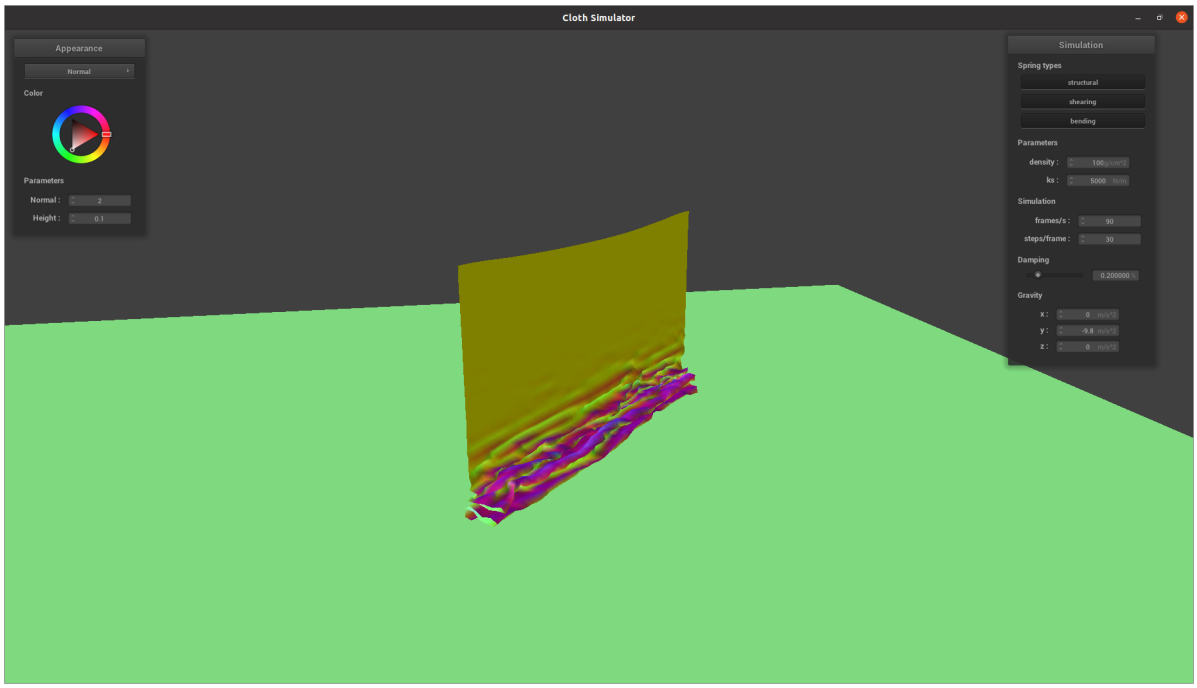


Low density cloth flattens out

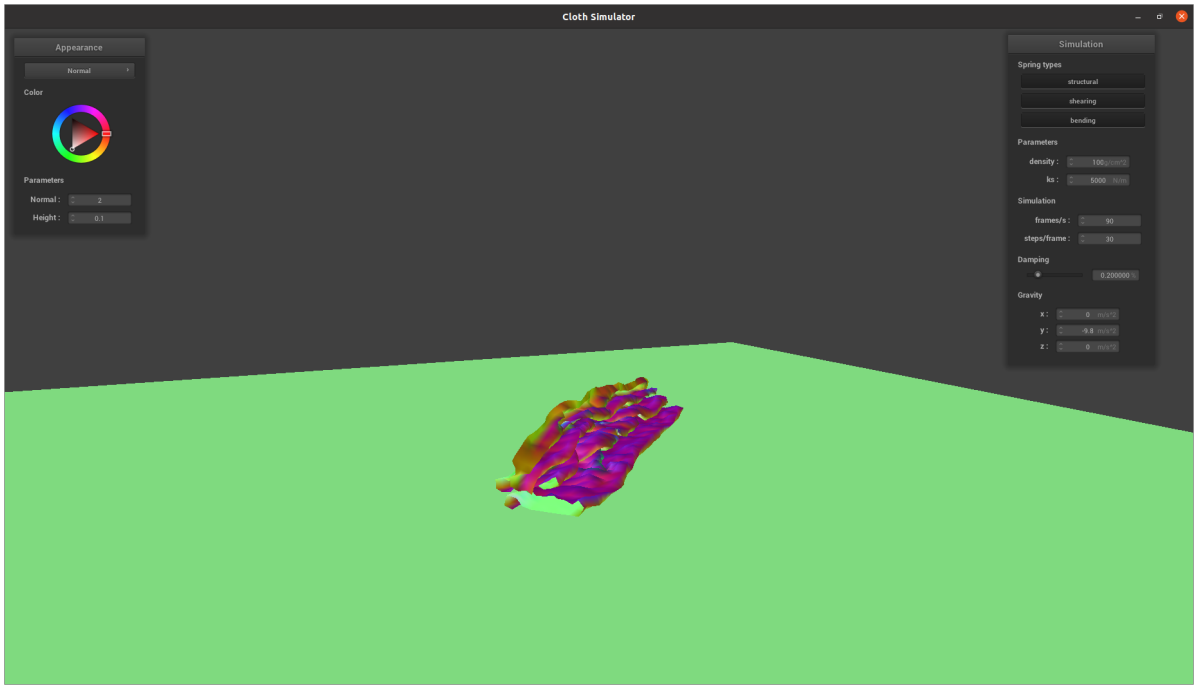


Low density cloth near rest state

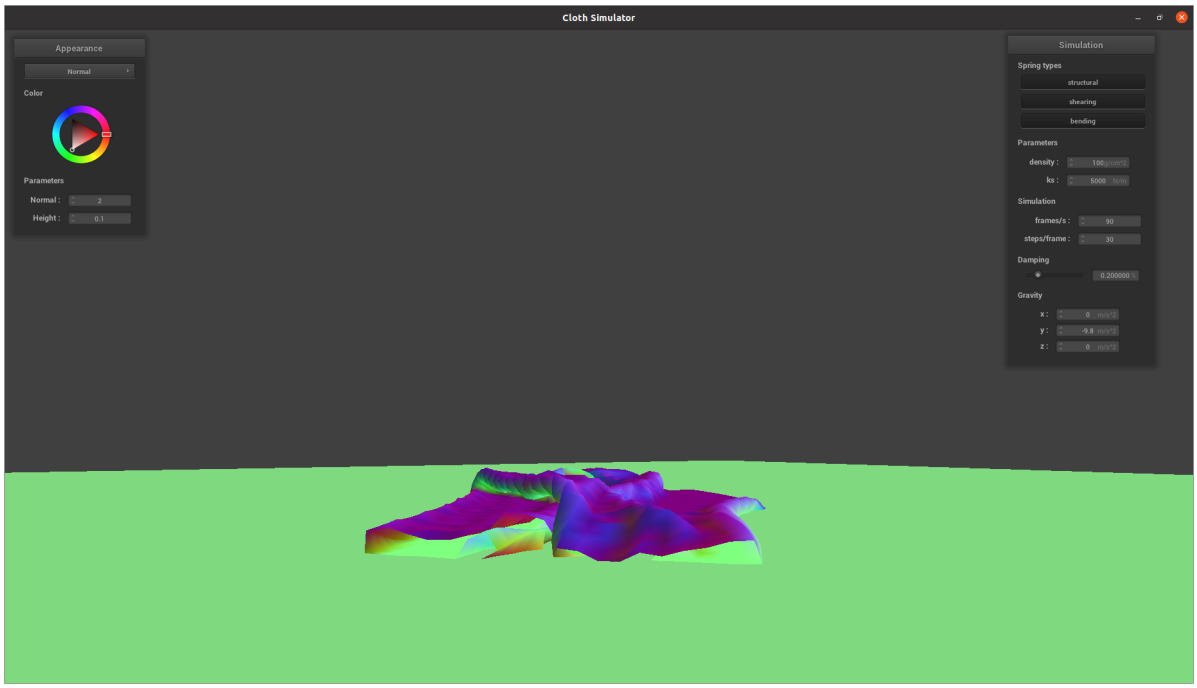
With a high density, a few thick waves slowly propagate through the cloth after its first self-collision. The cloth falls slowly and bounces less on itself when self-colliding. It continues to bounce quite a bit, but with less intensity, before becoming flat. When the cloth is flat, there are many thick fluid-like ripples that propagate throughout the cloth before it arrives at its resting state.



High density cloth begins to self collide

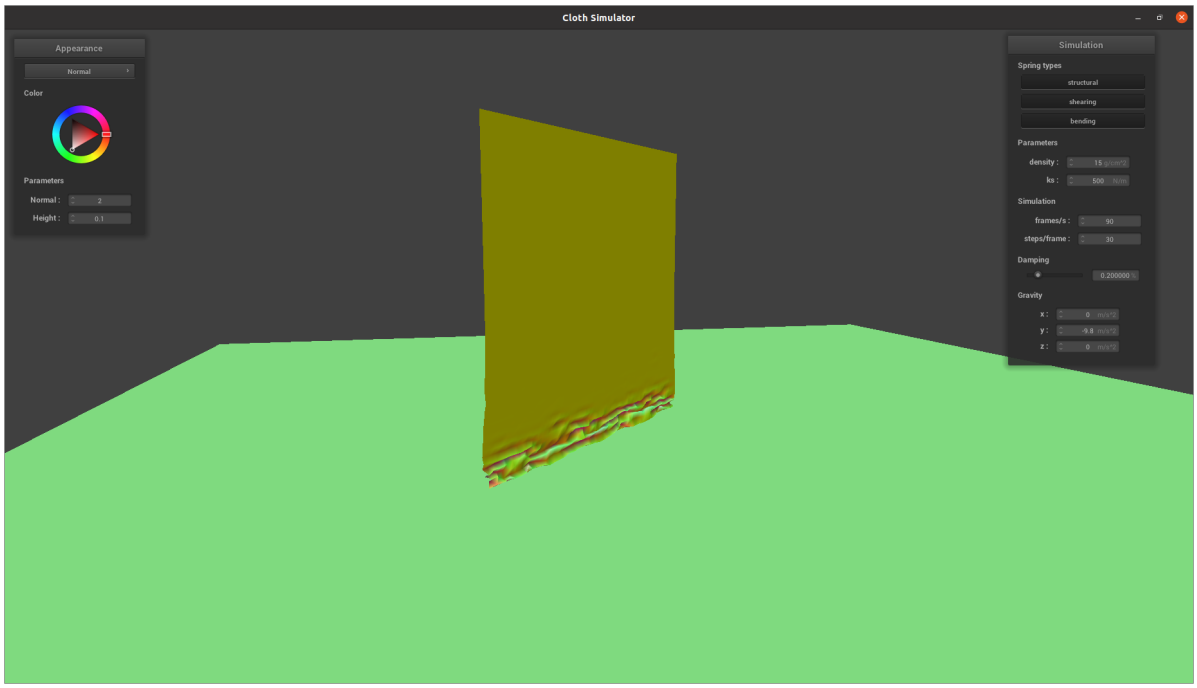


High density cloth falls to ground

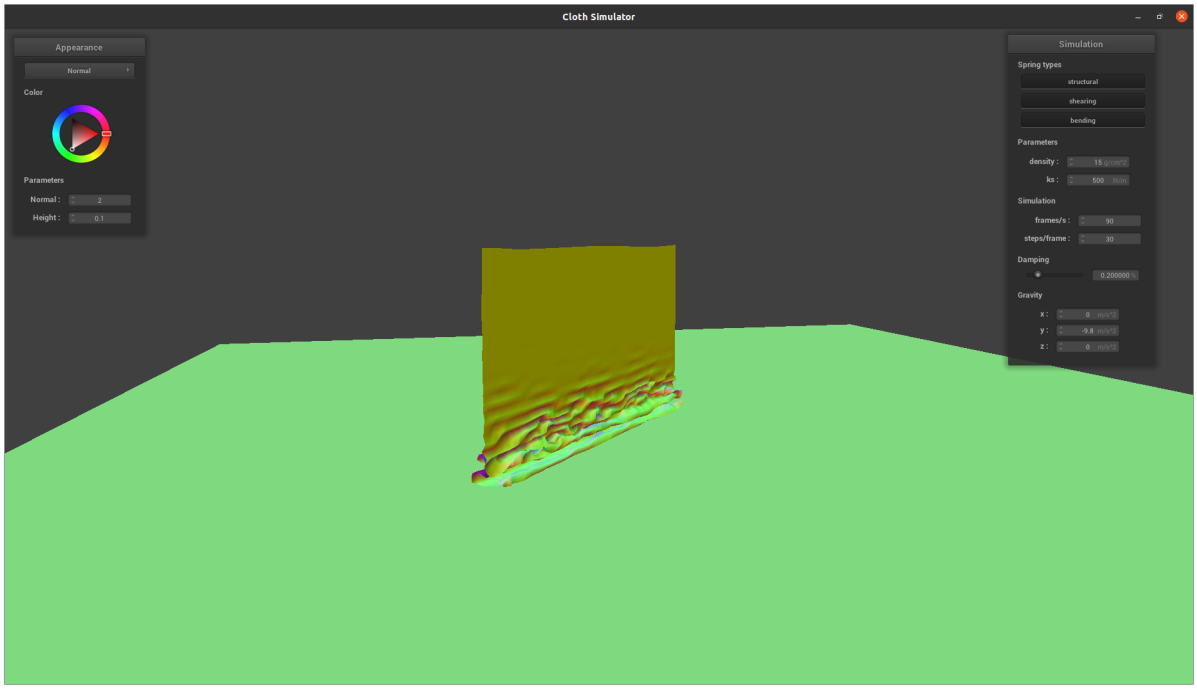


High density begins to flatten out

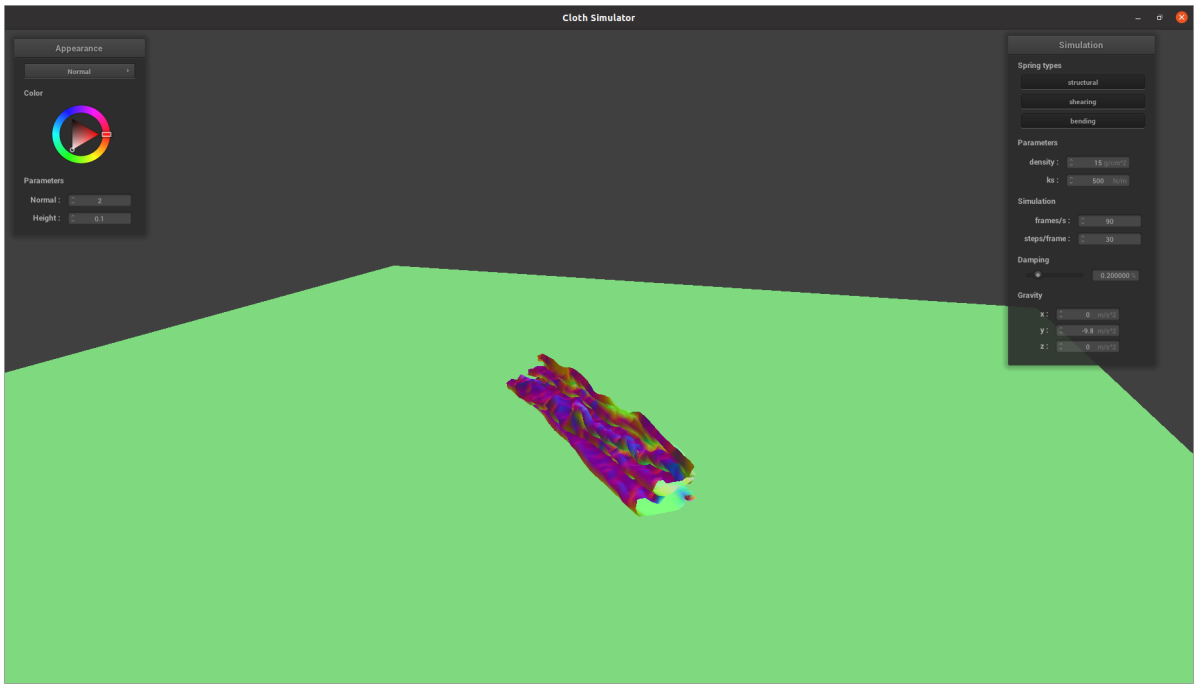
With a low spring constant, the cloth collisions are bouncy due to the spring-like cloth material and the folds are thin but plentiful. The cloth wrinkles and folds intensely upon multiple self-collisions during its fall. The cloth bounces on itself quite intensely with many small wrinkles and folds before flattening out like a sheet of jello.



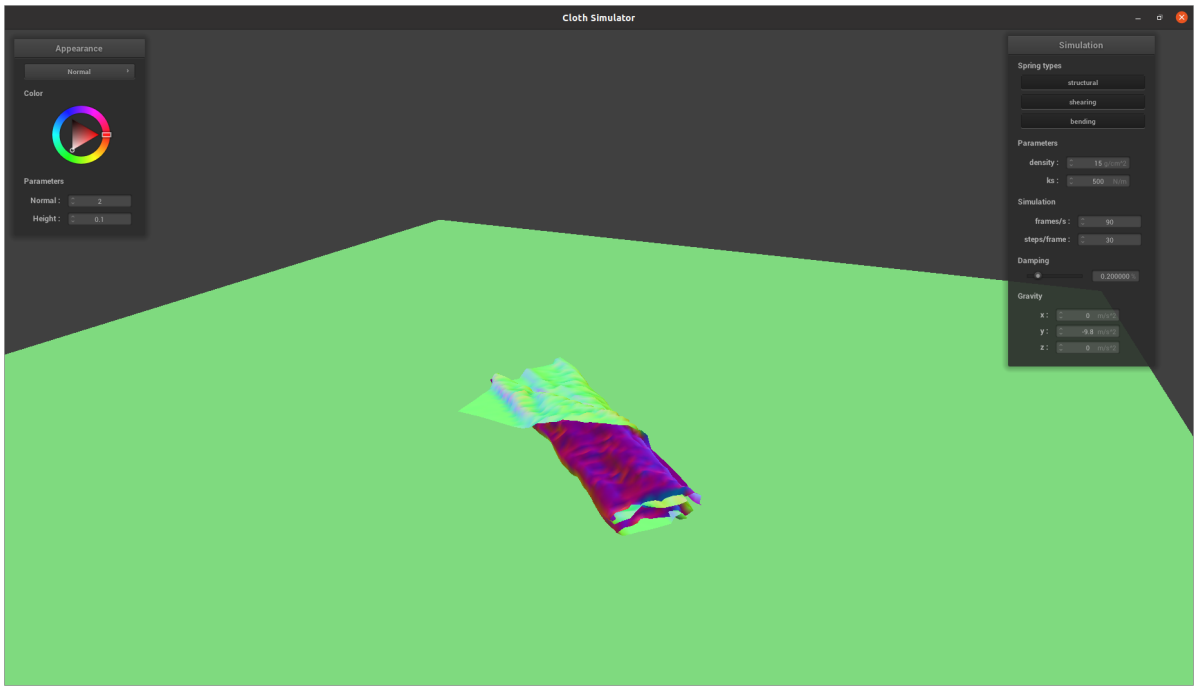
Low spring constant cloth begins to self-collide



Low spring constant cloth mid-fall

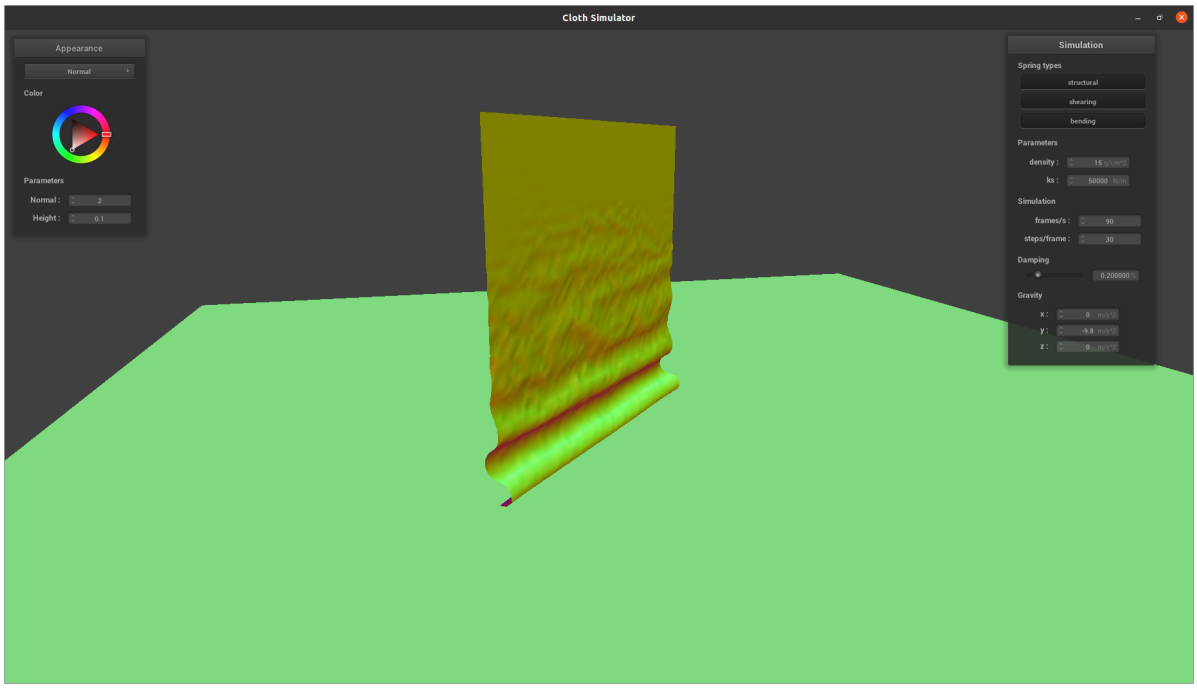


Low spring constant cloth falls to ground

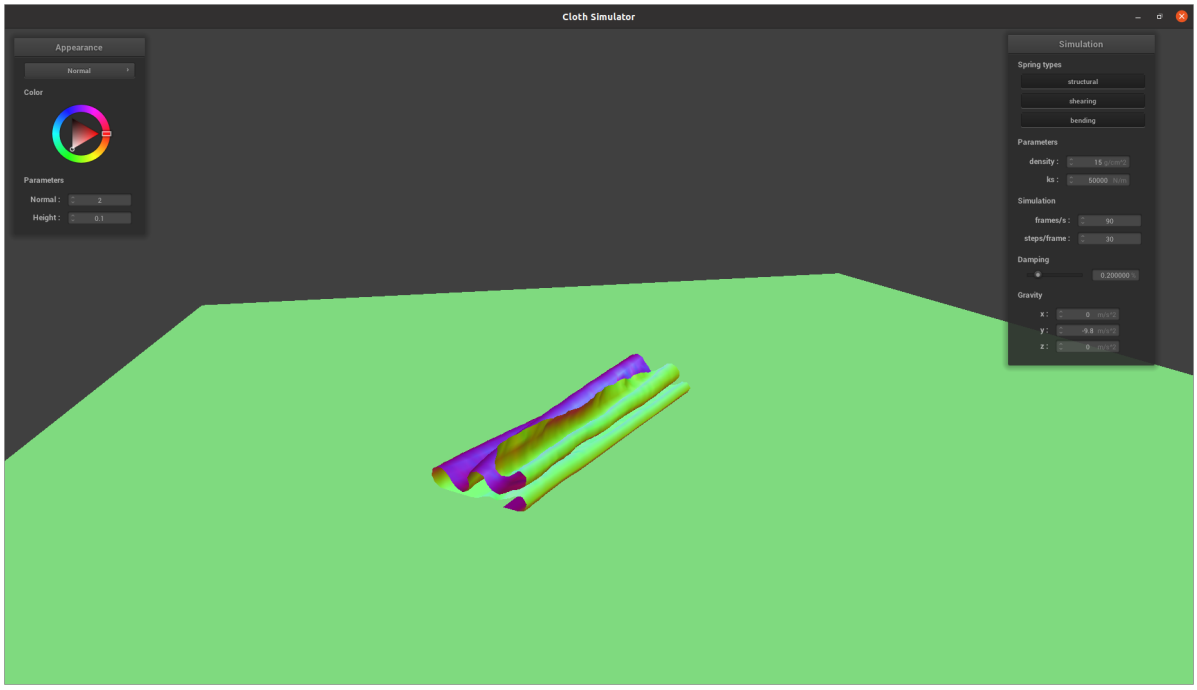


Low spring constant cloth at rest state

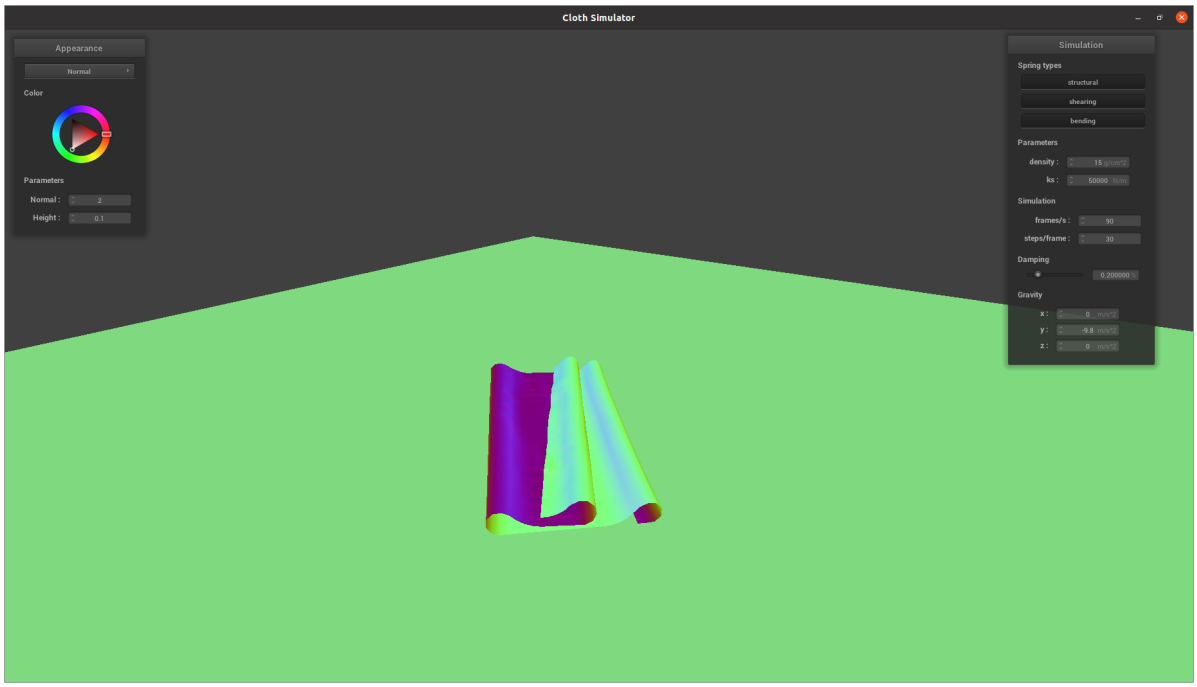
With a high spring constant, the cloth behaves like a stiff material. The cloth bounces less upon self collision and the folds are thick and fewer. The cloth flattens out quite quickly with fewer wrinkles and folds.



High spring constant cloth begins to self-collide



High spring constant cloth falls to ground



High spring constant cloth at rest state

Part 5

In this part I implement various shader vertex and fragment shaders, including the Blinn-Phong shading model, texture mapping, bump mapping, displacement mapping, and mirror material with environment-mapped reflections. I explain each of these implementations in further detail below.

- **Explain in your own words what is a shader program and how vertex and fragment shaders work together to create lighting and material effects.**

  A shader program is a program that is executed on the GPU and carries out a portion of the graphics pipeline and takes advantage of massive parallelism. Their massive performance speedup enables complex real-time 3D graphics.

  A GLSL shader is given inputs called *attributes* and global variables that are shared across each running instance of the shader called *uniforms* (allowing for parallelization). It outputs a single 4 dimensional vector and can write values into

varyings which can be used by a subsequent shader (e.g. a vertex shader passing information to a fragment shader).

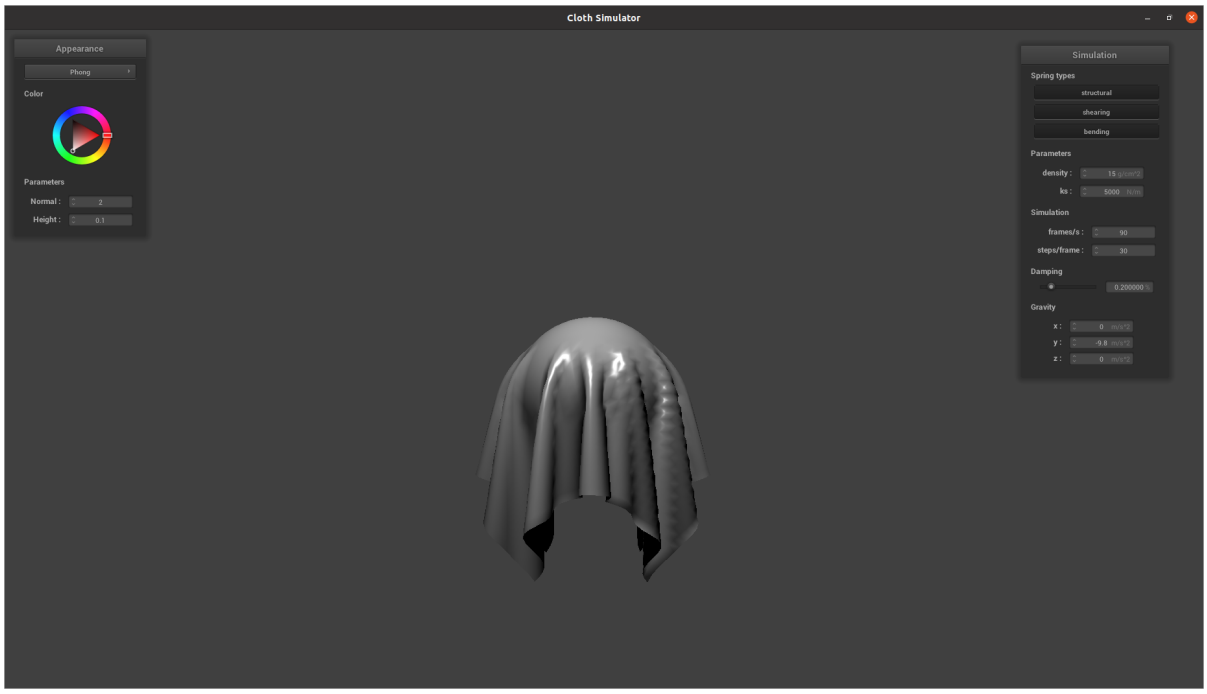
Shaders are typically used to create custom visual transformations such as complex shading and lighting, texturing, bump/displacement mapping, and simulate appearances leaves swaying in the wind or waves propagating through a fluid.

A vertex shader transforms the geometry of objects. It outputs the modified position of a vertex, and can also be used to transform other properties of the object geometry such as normal vectors. A vertex shaders outputs the final modified position of the vertex into `gl_Position` as a 4 dimensional vector (xyz and homogenous coordinate), and can write outputs into varyings for the fragment shader to use.

After the vertex shader has transformed the 3D geometry, the object is rasterized into 2D screen space, outputting *fragments*. These fragments can then be processed by the fragment shader.

A fragment shader transforms fragments, which are essentially abstract representation of pixels (although the comparison is not literal, as for example a pixel may sample from many fragments) and are the result of rasterization. Fragment shaders take in as inputs the properties of a fragment that is output by the vertex shader and performs a transformation. In GLSL, the final modified color value as a 4-dimensional rgba vector into `out_color`.

- **Explain the Blinn-Phong shading model in your own words. Show a screenshot of your Blinn-Phong shader outputting only the ambient component, a screen shot only outputting the diffuse component, a screen shot only outputting the specular component, and one using the entire Blinn-Phong model.**

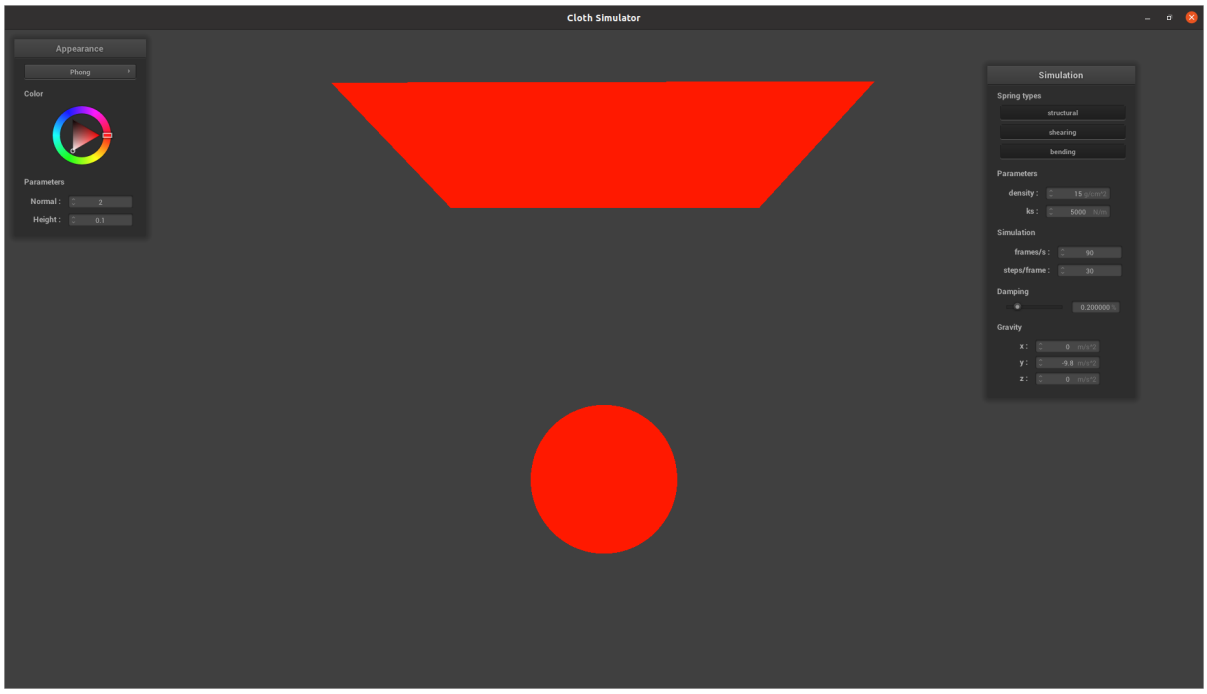


Showing rendering of Blinn-Phong shading model, with all 3 ambient, diffuse, and specular components.

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

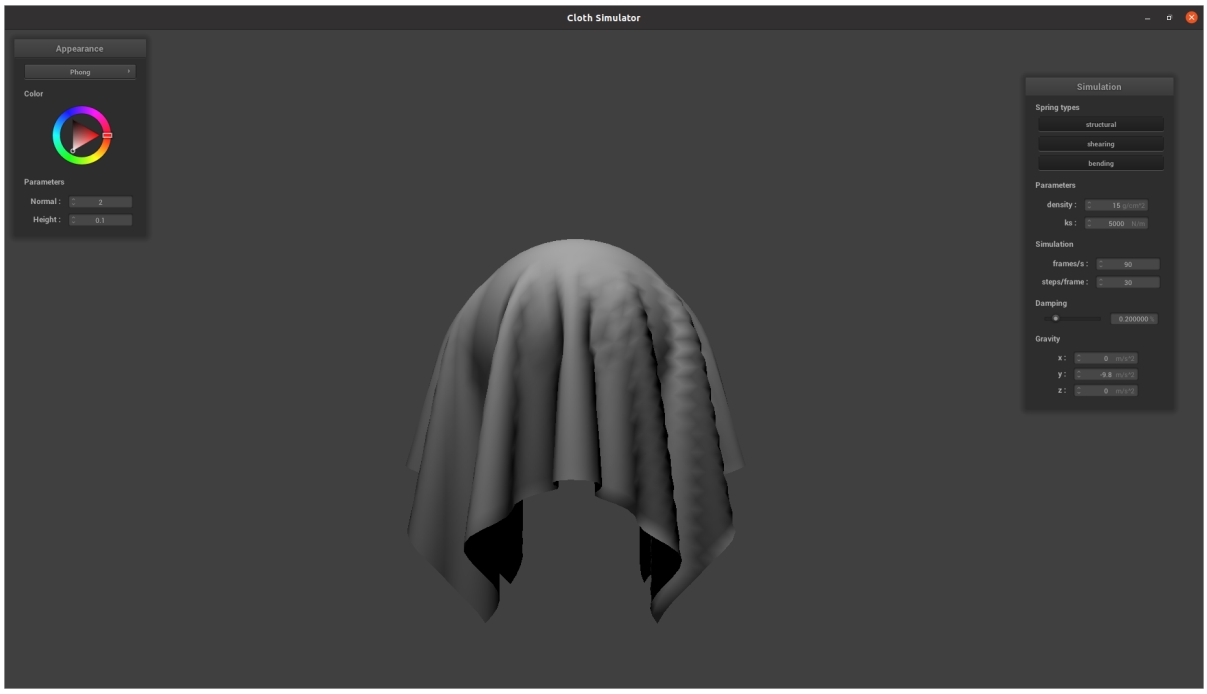
Blinn-Phong is a simple shading model that models 3 simplified components of lighting - ambient light, diffuse light and specular light.

Ambient light is a constant light that pervasive through the scene and is an extremely crude approximation to the accumulation of indirect light. The intensity and color of ambient light can be modified via the ambient coefficient and ambient intensity parameters, `k_a` and `I_a`.



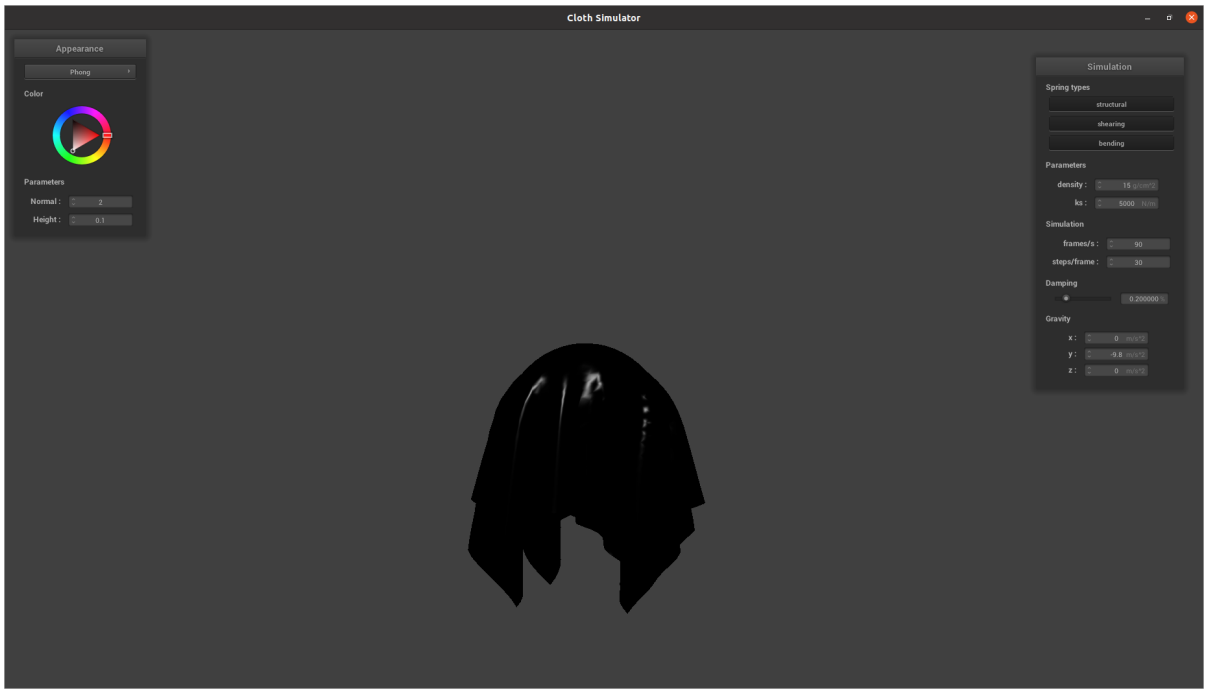
Showing only red ambient light

Diffuse light models the scattering of light off of the surface an object, and different levels of scattering can be specified via the diffuse coefficient `k_d`. Diffuse light takes into the account the distance squared fall off of the light intensity from a light source. Diffuse light is dependent on the orientation of the surface and the direction of incoming light. and is independent of view direction.



Showing only diffuse light

Specular light models the concentrated reflection of flight that results from shiny and reflective surfaces. Specular light is dependent on view direction, surface orientation, and light direction (in the equation above, h is defined as the bisector of the normalized view and light vectors in the direction away from the surface). The concentration of the specular light can be modified through the the specular coefficient k_s and albedo p, which is a measure of a material's roughness.

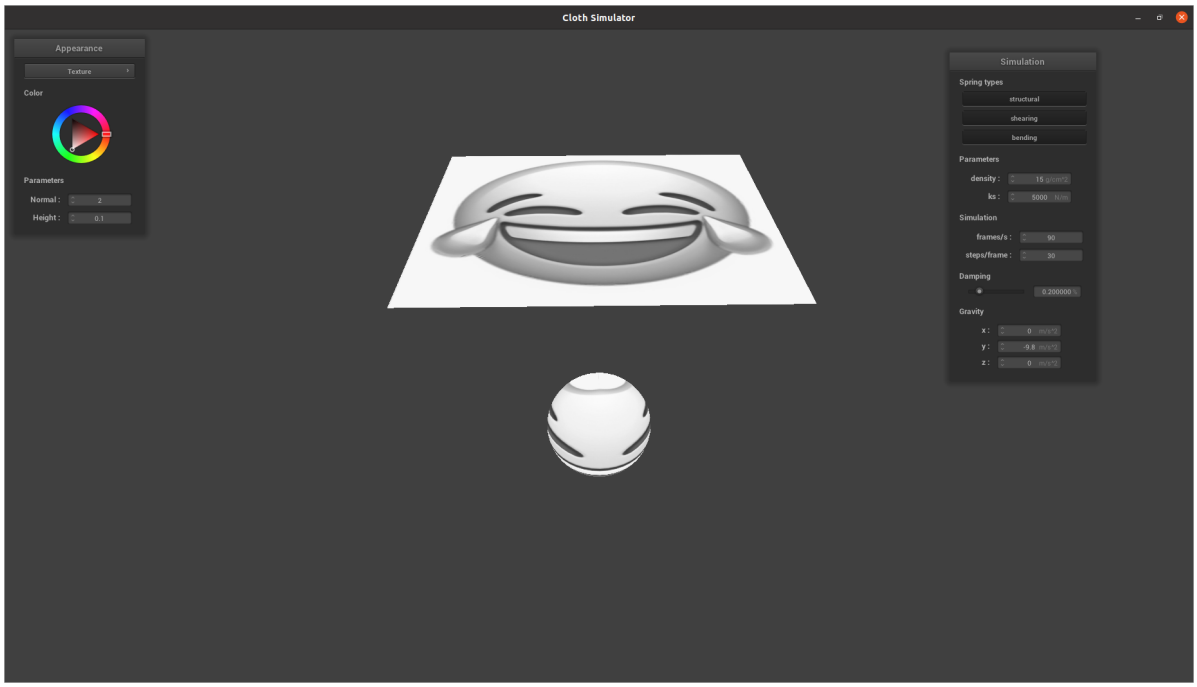


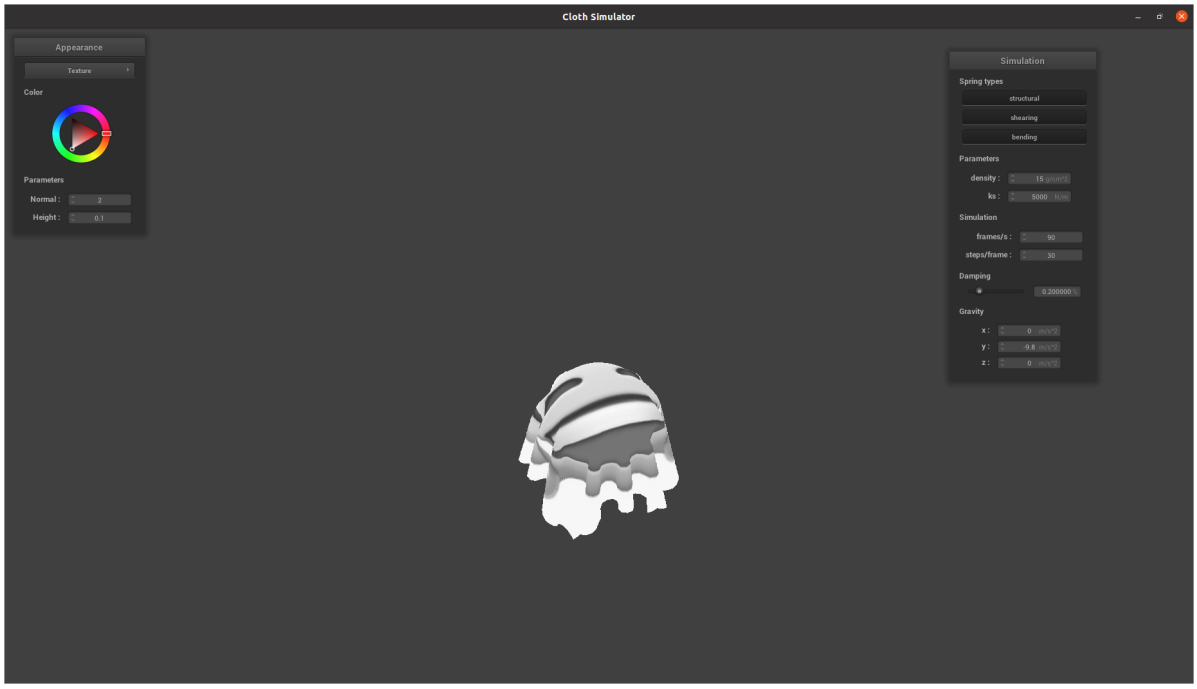
Showing only specular light

- **Show a screenshot of your texture mapping shader using your own custom texture by modifying the textures in `/textures/`.**

  I implemented texture mapping in a fragment shader by setting the `out_color` to be the result of calling the GLSL `texture` function with the texture and uv-texture coordinates.

  I put the Apple crying laughing emoji as one of the textures.





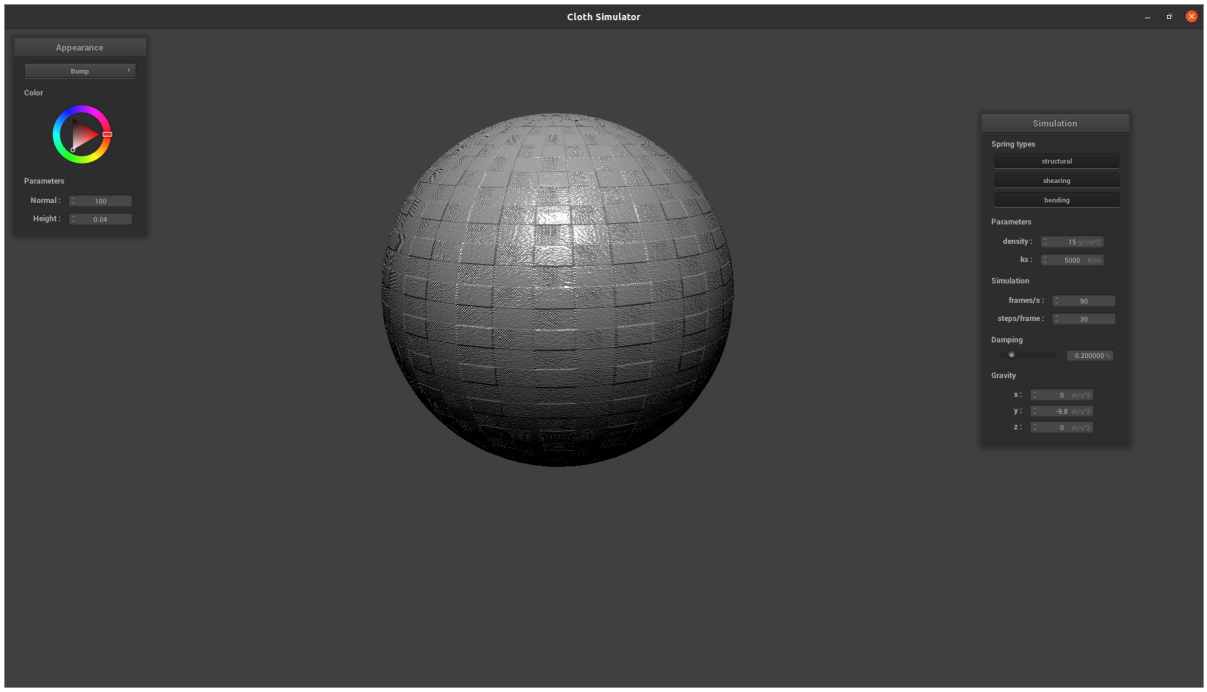
- **Show a screenshot of bump mapping on the cloth and on the sphere. Show a screenshot of displacement mapping on the sphere. Use the same texture for both renders. You can either provide your own texture or use one of the ones in the textures directory, BUT choose one that's not the default `texture_2.png`. Compare the two approaches and resulting renders in your own words.**

Compare how your the two shaders react to the sphere by changing the sphere mesh's coarseness by using `o 16 -a 16` and then `o 128 -a 128`.

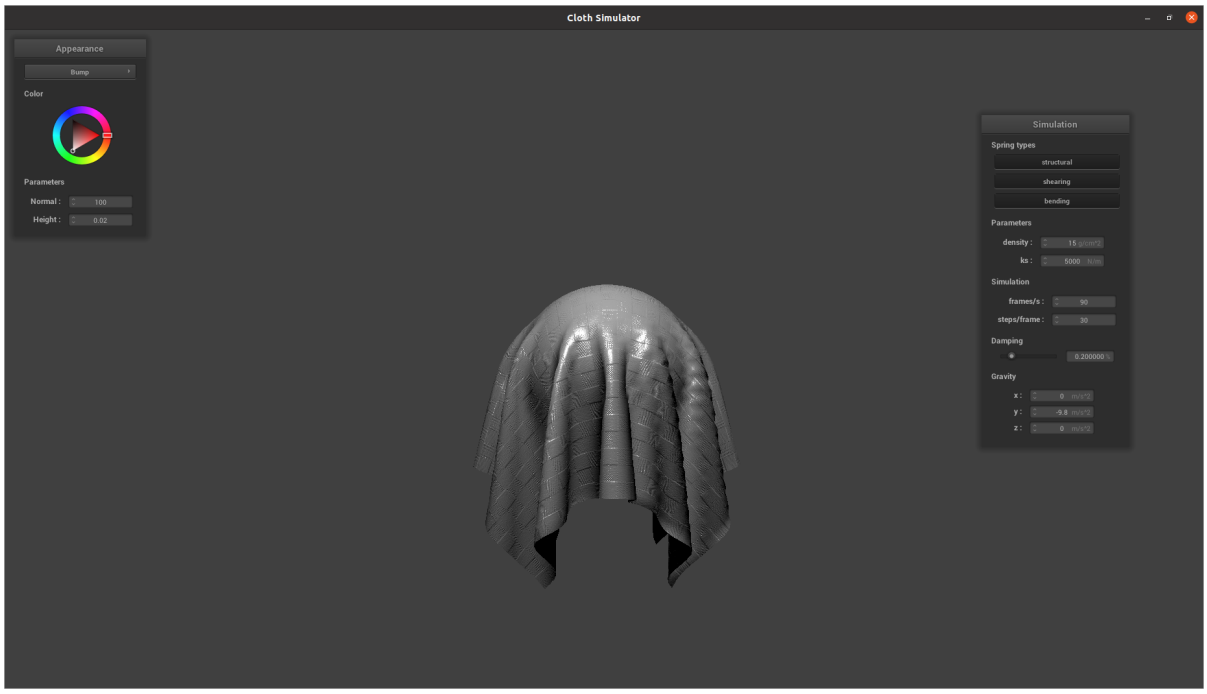
Both bump mapping and displacement mapping are methods of creating the appearance of texture.

Bump mapping creates the illusion of texture by modifying the surface normal via a height map specified by a texture, modifying the way light is reflected and calculated. However, the vertex positions themselves are not modified, so the appearance of texture is less convincing.

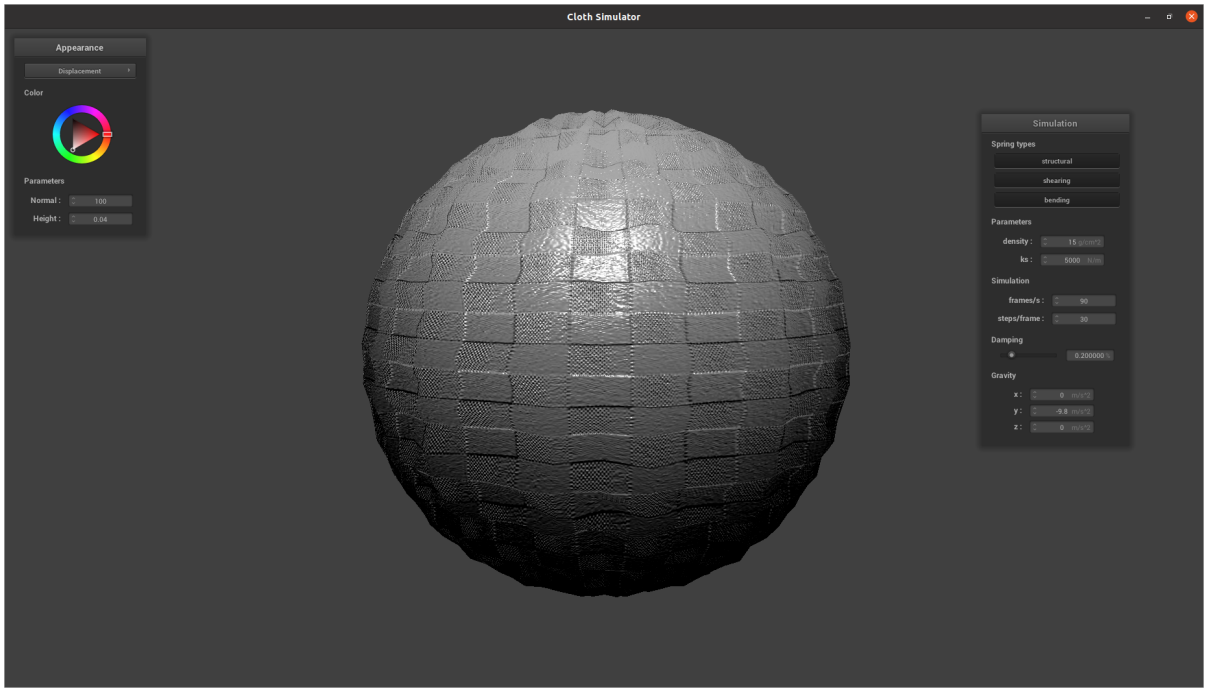
Displacement creates the appearance of texture by actually modifying the output position of the vertices via a height map specified by a texture. This leads to a more visible and realistic simulation of texture, where you can see the vertices physically displaced from their original location.



Bump mapping

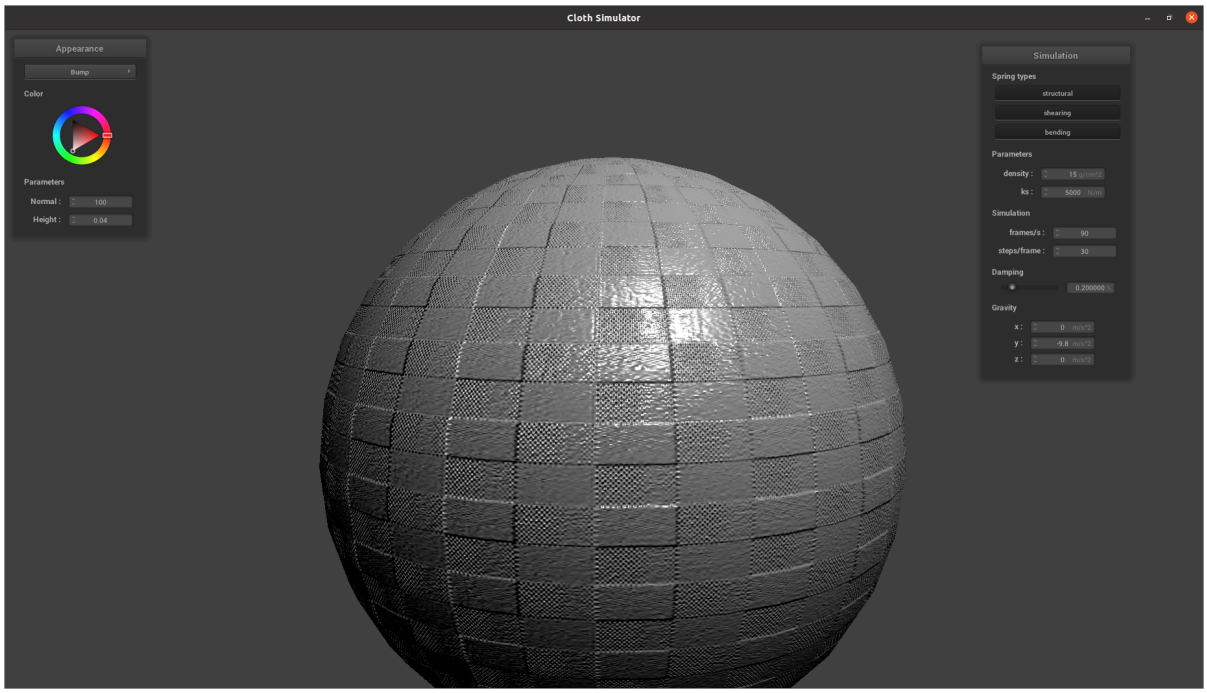


Bump mapping

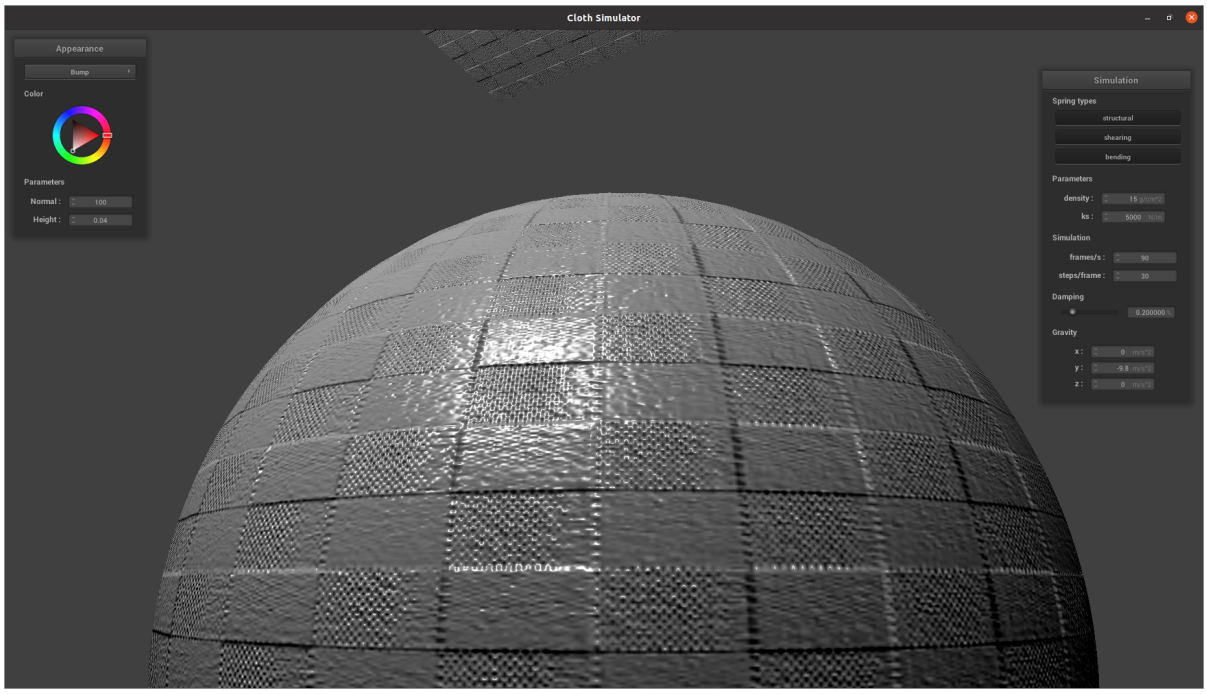


Displacement mapping

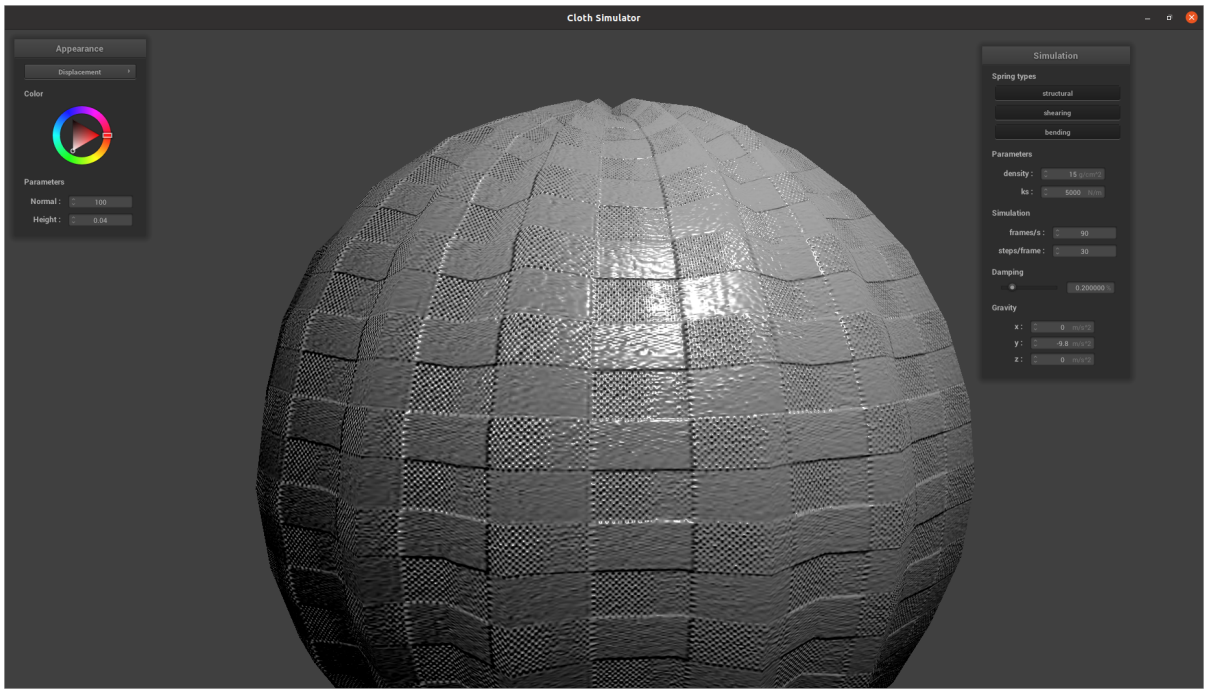
With a small coarseness of 16, both bump mapping and displacement mapping are essentially equivalent since the magnitude of the divets is very small - creating the illusion of coarseness by modifying normals in normal mapping is sufficient. However with a heavy coarseness of 128, displacement mapping is much more accurate since the magnitude of the divets is large and we can see the physical geometric distortion in actually displacing the position of the vertices in displacement mapping.



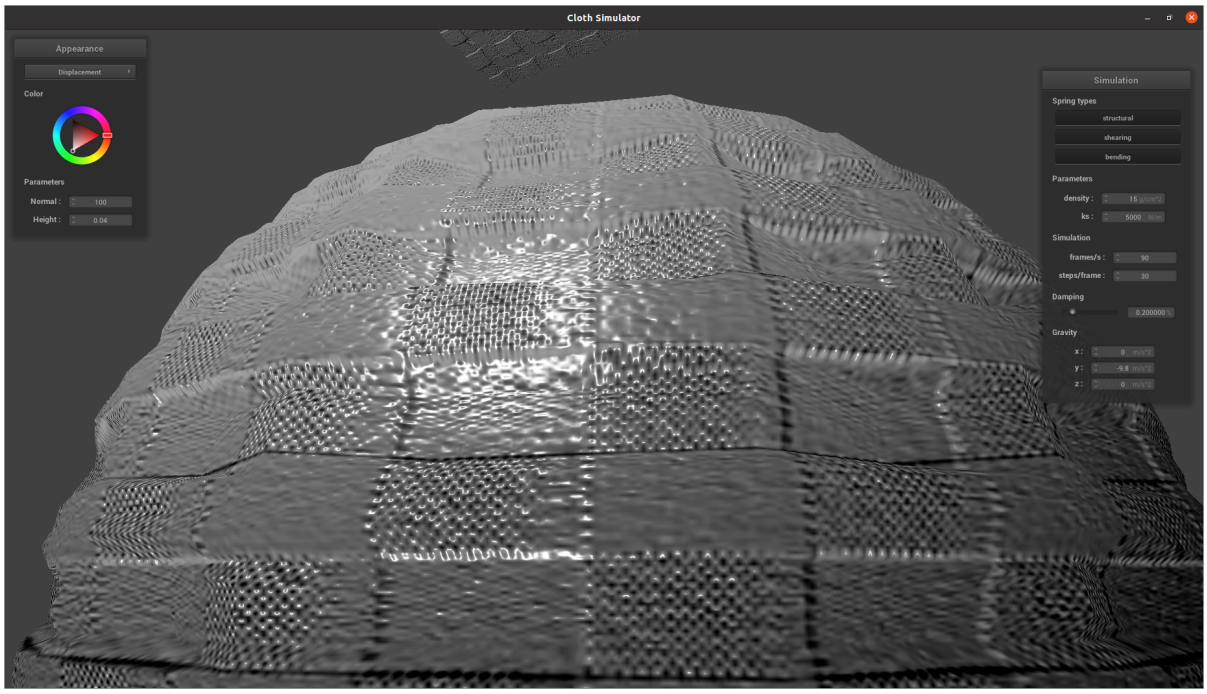
Bump mapping with coarseness `o 16 -a 16`



Bump mapping with coarseness `o 128 -a 128`



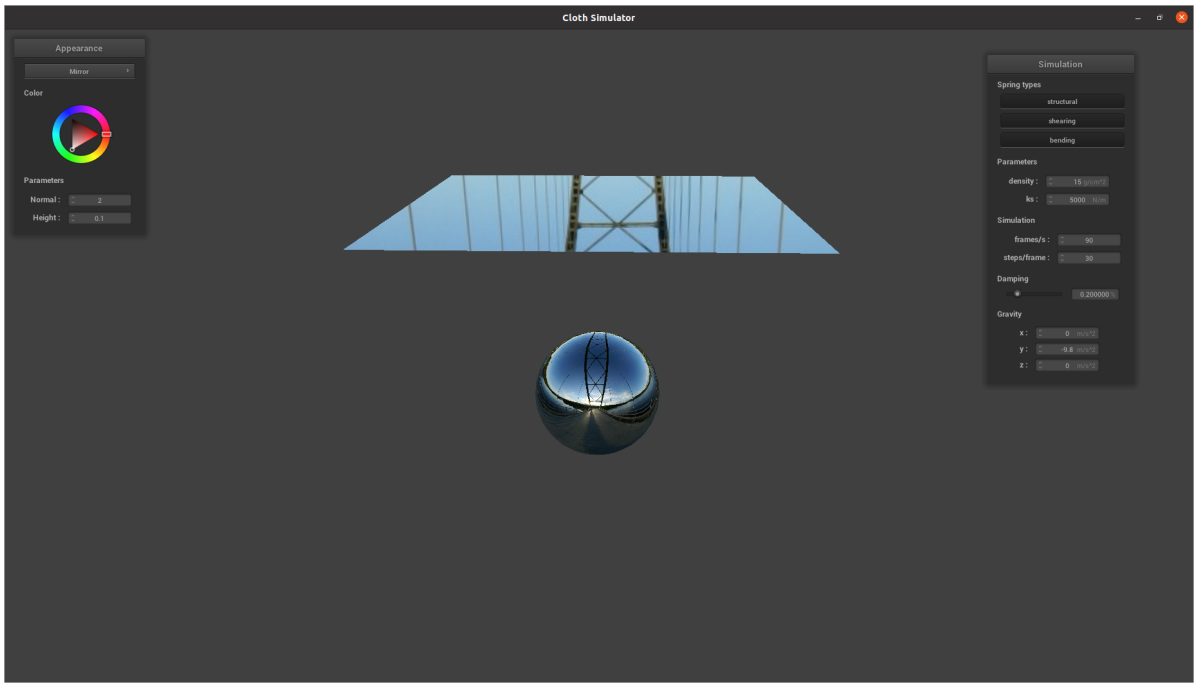
Displacement mapping with coarseness `o 16 -a 16`

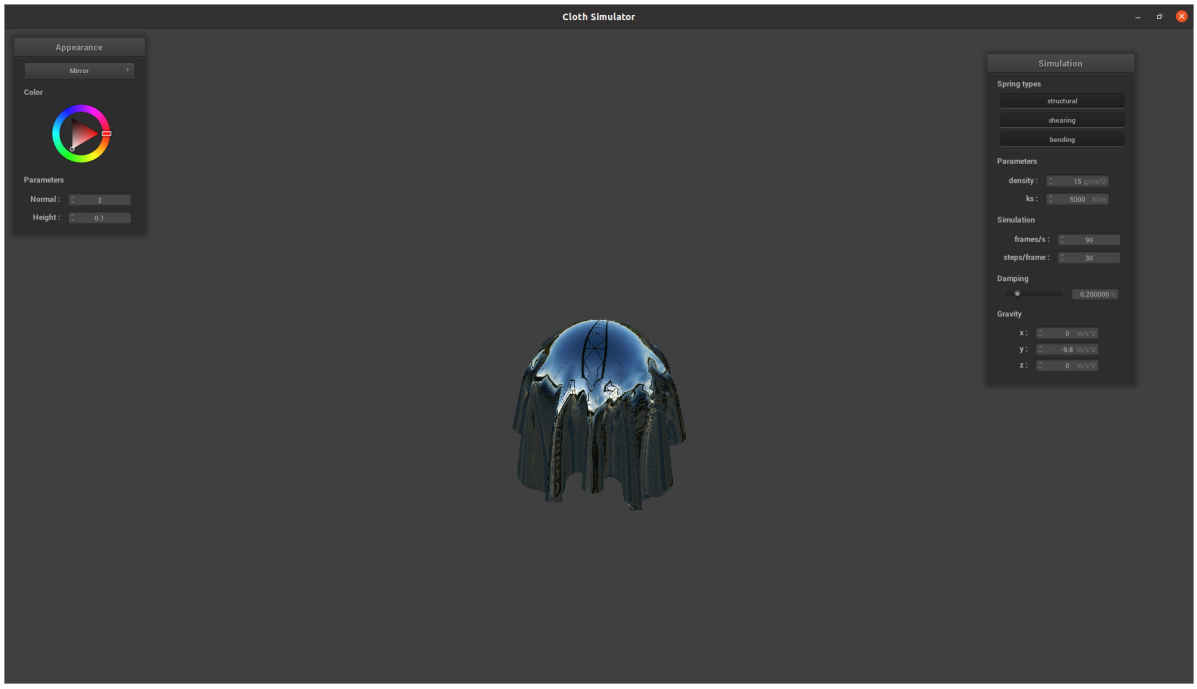


Displacement mapping with coarseness `o 128 -a 128`

- **Show a screenshot of your mirror shader on the cloth and on the sphere.**

  To implement the mirror fragment shader, I compute the normalized view vector from the camera to the position. Then, I reflect the view vector across the normal using the GLSL `reflect` function. I then set `out_color` to the result of sampling the texture cubemap, similar to texture mapping, where I call the GLSL `texture` function with the cubemap and outgoing reflected direction.


Cloth Simulator
Appearance
Mirror
Color
Parameters
Normal :
Height :
Simulation
Spring types
structural
shearing
bending
Parameters
density :
ks :
Simulation
frames/s :
steps/frame :
Damping
Gravity
x :
y :
z :


Cloth Simulator
Appearance
Mirror
Color
Parameters
Normal :
Height :
Simulation
Spring types
structural
shearing
bending
Parameters
density :
ks :
Simulation
frames/s :
steps/frame :
Damping
Gravity
x :
y :
z :