Cloth Sim

by Jonah Bedouch and Brandon Wong

2025-04-07

This report is also available online at https://cs184-zen.vercel.app/projects/hw04.

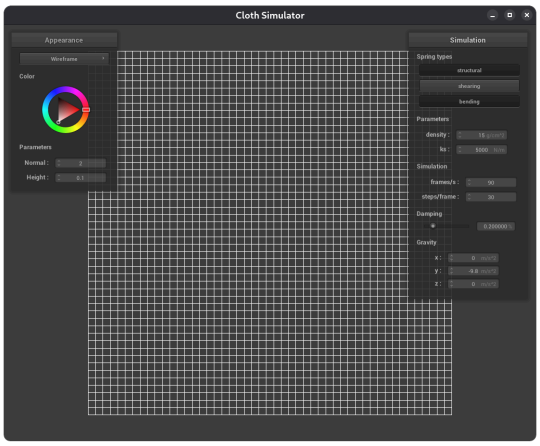
The GitHub repository for this project can be found at
https://github.com/cal-cs184-student/sp25-hw4-crashing-out/

Overview

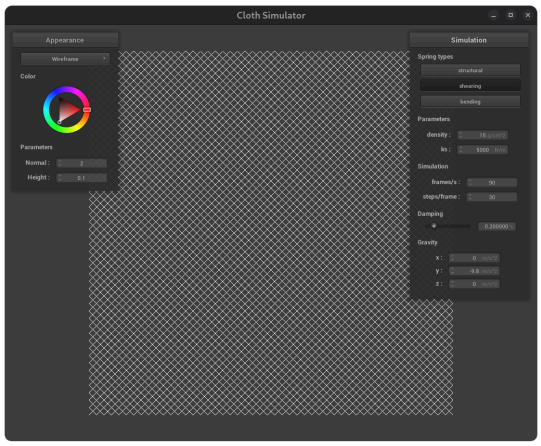
In this homework, we implement a cloth simulation. We begin by creating our cloth itself, which is represented by a system of point masses connected together via springs. These springs enforce constraints on where the point masses can move, allowing us to simulate the elasticity of a cloth while maintaining shape. Then, we actually simulate the physics behind this point mass system through numerical integration, implementing the Verlet numerical integrator to allow us to simulate a system where gravity is imposed on this cloth, moving points based off of all of the constraints imposed by the spring-mass system. From there, we introduce other physical objects, like a sphere and a plane, and implement collisions with these objects by essentially applying a normal force when a point crossese under the threshold into the sphere/plane. This force counteracts the downwards force of gravity, forcing the cloth to stay on the sphere/plane. After this, we add self-collisions, so that the cloth realistically collapses onto itself rather than clipping through itself. These collisions are implemented as a minimum spring size that is enforced by applying a correcting force on any two points that are closer together than this minimum. Finally, after our system is physically accurate, we use GLSL to implement shaders that allow us to realistically shade our cloth, with many different material options – diffuse, phong, mirror, and so on. This results in a somewhat accurate, pleasant looking cloth simulation.

Part 1: Masses and Springs

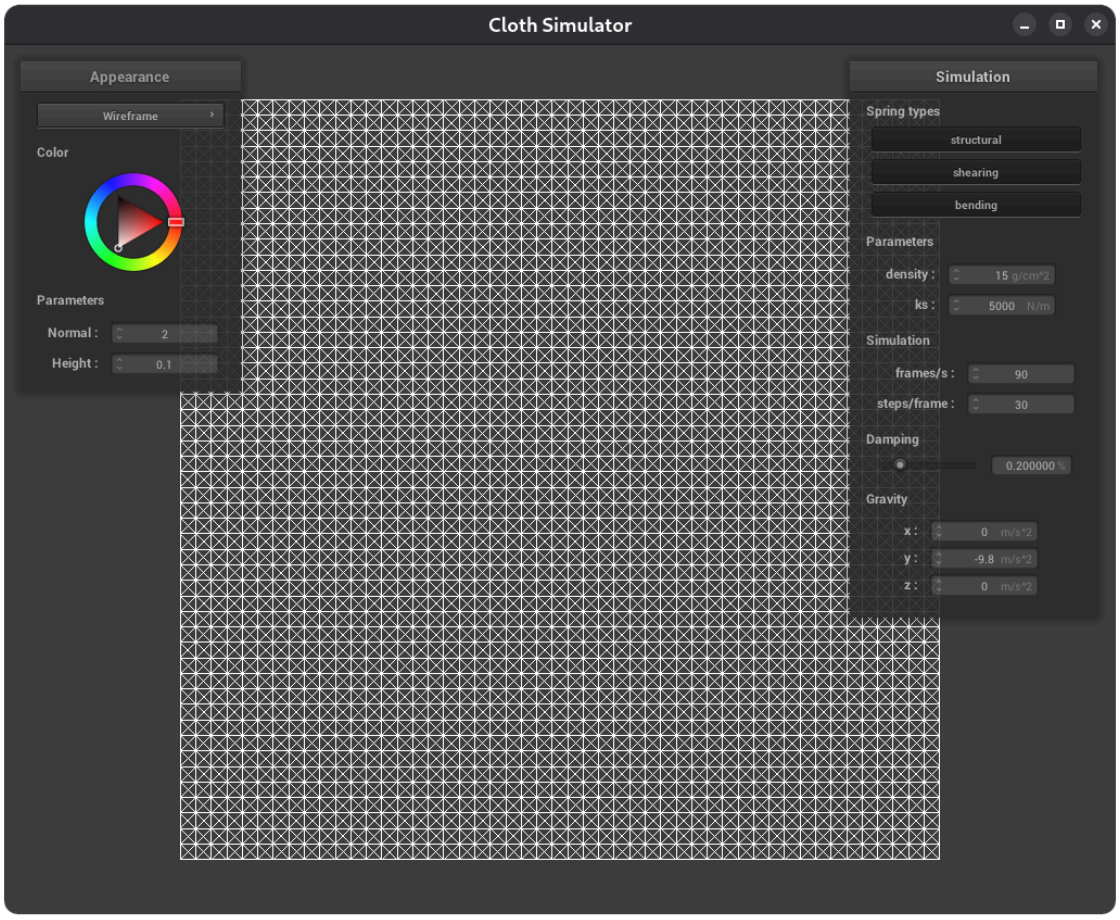
For this part, we created all of our points using a dual for loop and placed them in the point masses vector. We determined the positions by finding offsets based on the width and height divided by the number of points minus one (to account for the fact that one set of points would be at 0,0), then multiplying them by the current x or y index. After this, we looped through each point again to place the springs, placing the necessary structural (left and above), shearing (upper left and upper right), and bending (two to the left and two to the right) springs for each point. If statements were used to account for edge cases to ensure springs whose second point would have been out of bounds were not created.



`pinned2.json` , shearing springs hidden



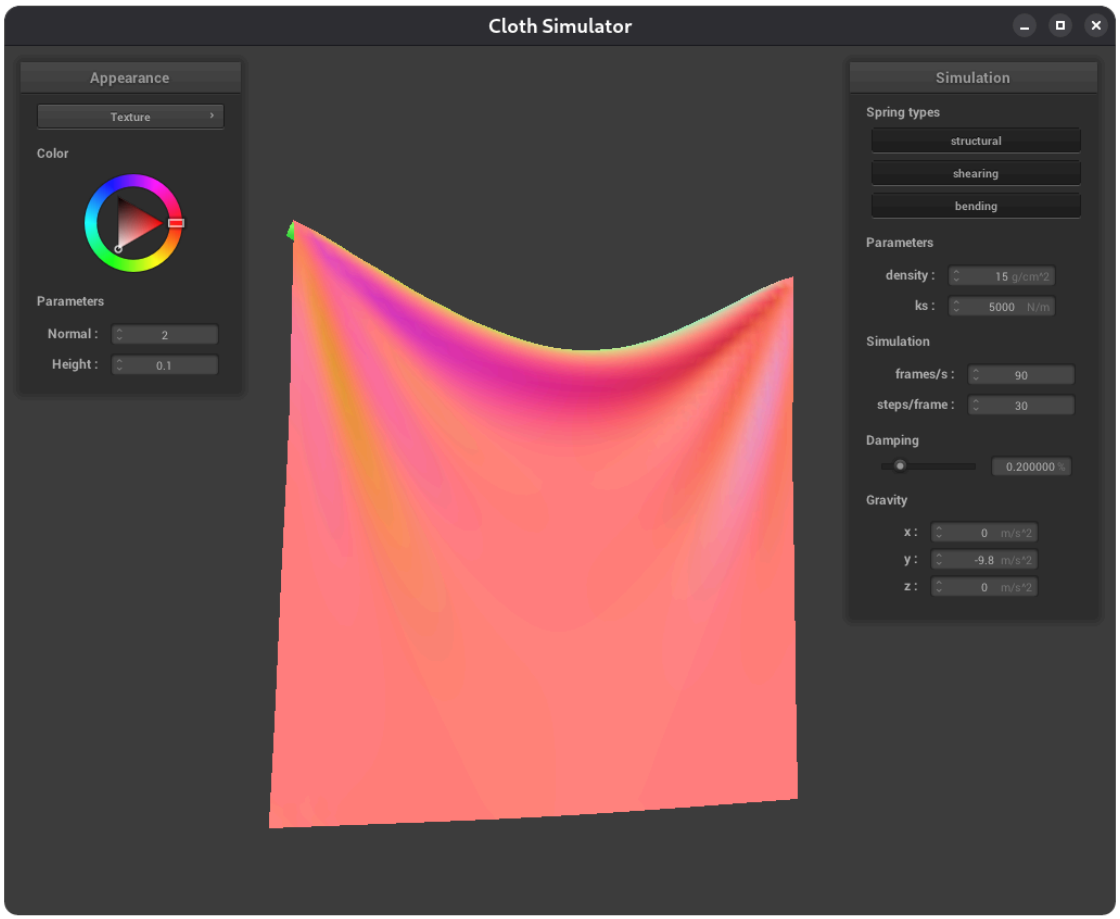
`pinned2.json` , only shearing springs



`pinned2.json`, all springs shown

Part 2: Simulation via Numerical Integration

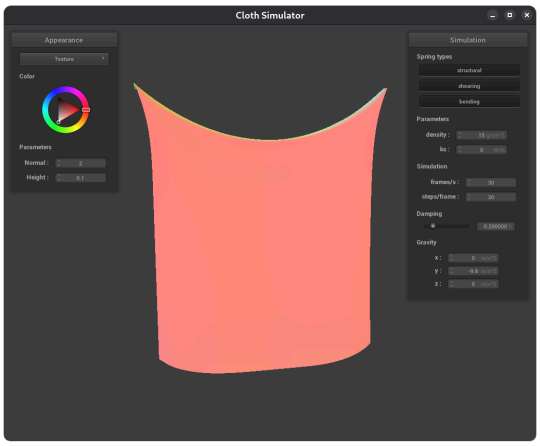
Throughout this part, we implemented the Verlet numerical integrator to allow for the cloth to actually move as a result of external forces and the forces of the springs that we implemented in part 1. Below is a picture of a cloth that has reached steady state.



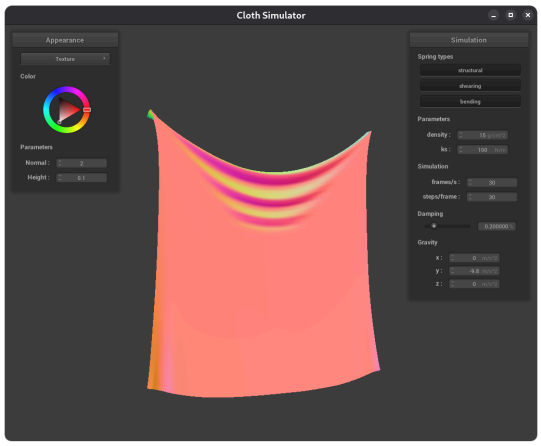
`pinned4.json` at steady state with default parameters

Changing the Spring Constant ks

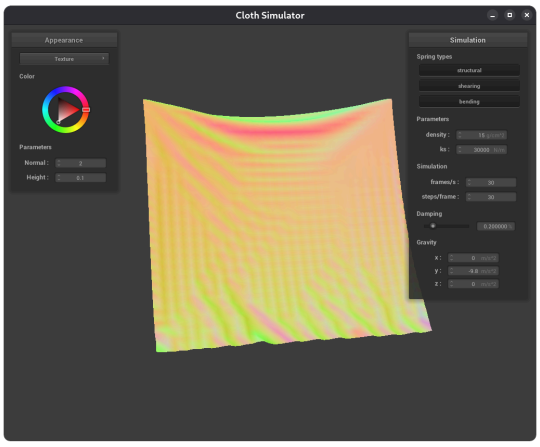
At low ks, the cloth appears to become smoother. At, zero, there is no force from the spring so the cloth falls flat and has no present wrinkles. This flatness decreases as the ks goes up, eventually reaching a point where the ks causes the spring force to surpass gravity. When this occurs, the resulting cloth becomes very wavy as the force of the springs causes it to oscillate, seemingly never reaching a solid steady state. In between, there is a point at which ks is high enough that visible wrinkles quickly develop along the entire cloth as it falls, and never disappear.



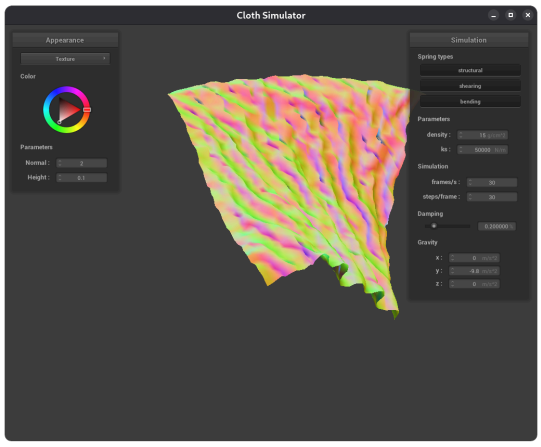
`pinned4.json` with ks=0



`pinned4.json` with ks=100



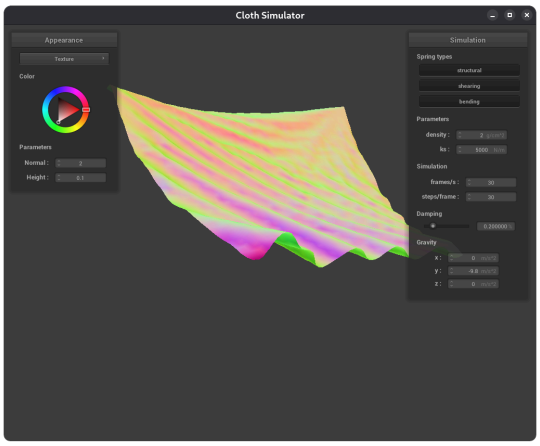
`pinned4.json` with ks=30000



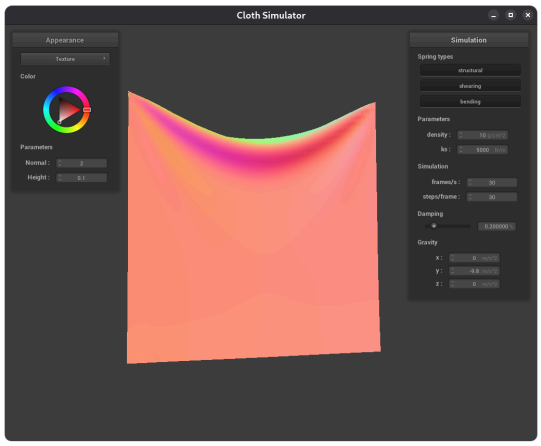
`pinned4.json` with ks=50000

Changing the Density

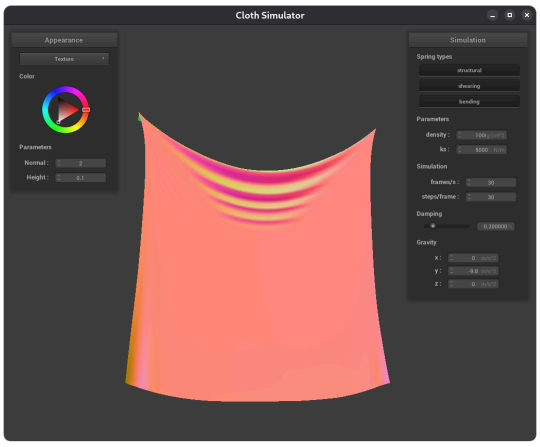
Very low density resulted in an effect very similar to high ks, where the low density caused the spring force to seemeingly overpower the force from gravity, resulting in sinusoidal results in the cloth. Slightly higher but low density had generally smooth results, but differed from the standard density by having less pronounced wrinkles on the top. Medium high density had more of the wrinkles at the center of the top. Finally, high density resulted in a cloth that was very flat, as the higher mass meant the force from the springs could not significantly change the effect from gravity. In general, the higher the density, stronger the force of gravity was relative to the force of the spring constants, so lower density allowed the impacts of the springs to be more pronounced, while a high enough density completely overpowered them.



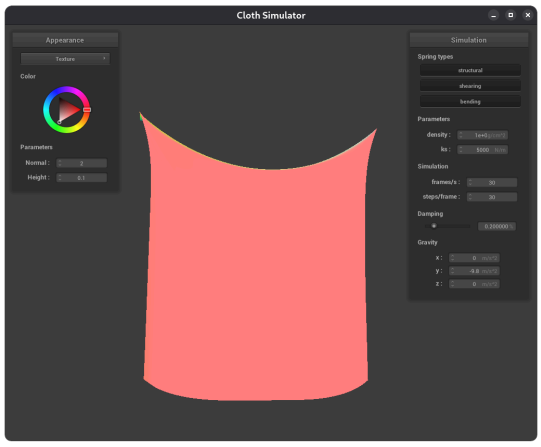
`pinned4.json` with density=2



`pinned4.json` with density=10



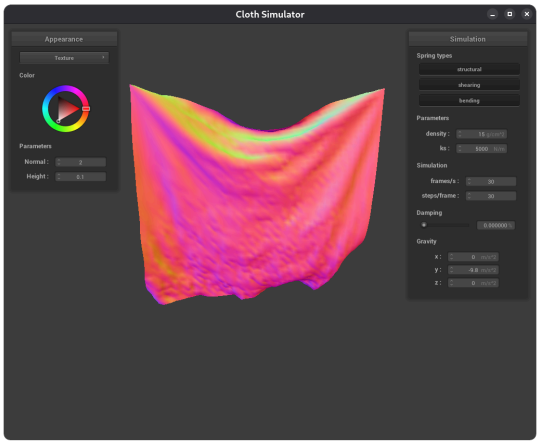
`pinned4.json` with density=1000



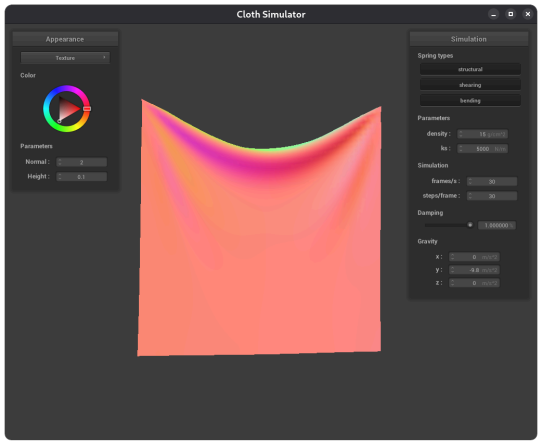
`pinned4.json` with density=1e+7

Changing the Damping

Low damping caused the entire cloth to very quickly flutter, as though a strong gust of wind was being applied in one direction, then flipping to be applied in the other. This happened very quickly, and repeated with the entire cloth oscillating back and forth and being held together by the two fixed points. Because the damping effect was 0, there was nothing to cause the force on the spring to decrease, so the points continually had new force applied to them causing the constant motion. High damping caused the cloth to have very slow changes, as damping limited the effects of the forces on it. It took a long time to reach the steady state. The fall and final steady state of high damping looked very similar to the default damping, with very slightly less pronounced wrinkles, but it took longer to get there.

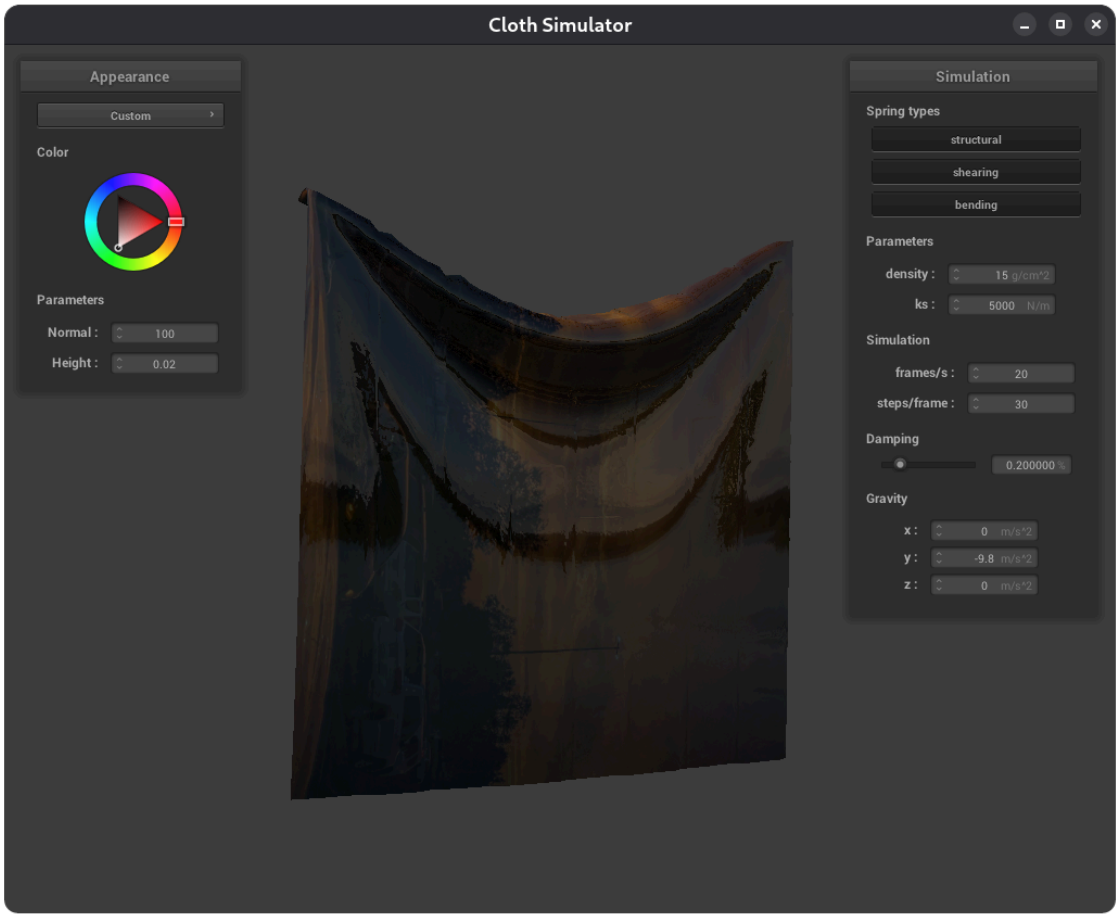


`pinned4.json` with damping=0



`pinned4.json` with damping=1

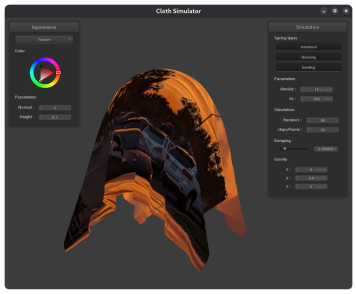
After implementing shading and getting everything working, we get a final product of:



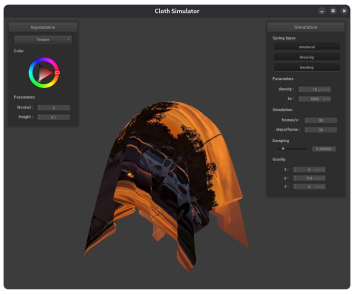
`pinned4.json`, shaded in final resting state

Part 3: Handling Collisions with Other Objects

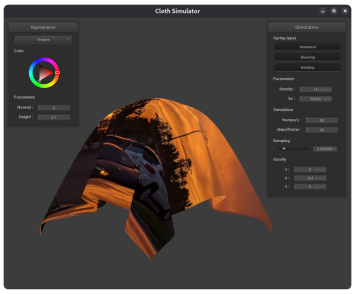
To implement sphere collisions, we find the vector pointing from the origin to the position (by subtracting the origin from the position), and then normalize it to get the magnitude. If the magnitude is less than the radius, we know that we have collided, and need to adjust the position. To find our adjusted position, we find the unit vector between the origin and position, multiply it by the radius, and then add the origin (to translate from the origin to the point on the sphere along the vector between the two points). Then, we find a correction vector by simply subtracting the last position from the adjusted position, and apply it as described in the spec.



`ks=500`

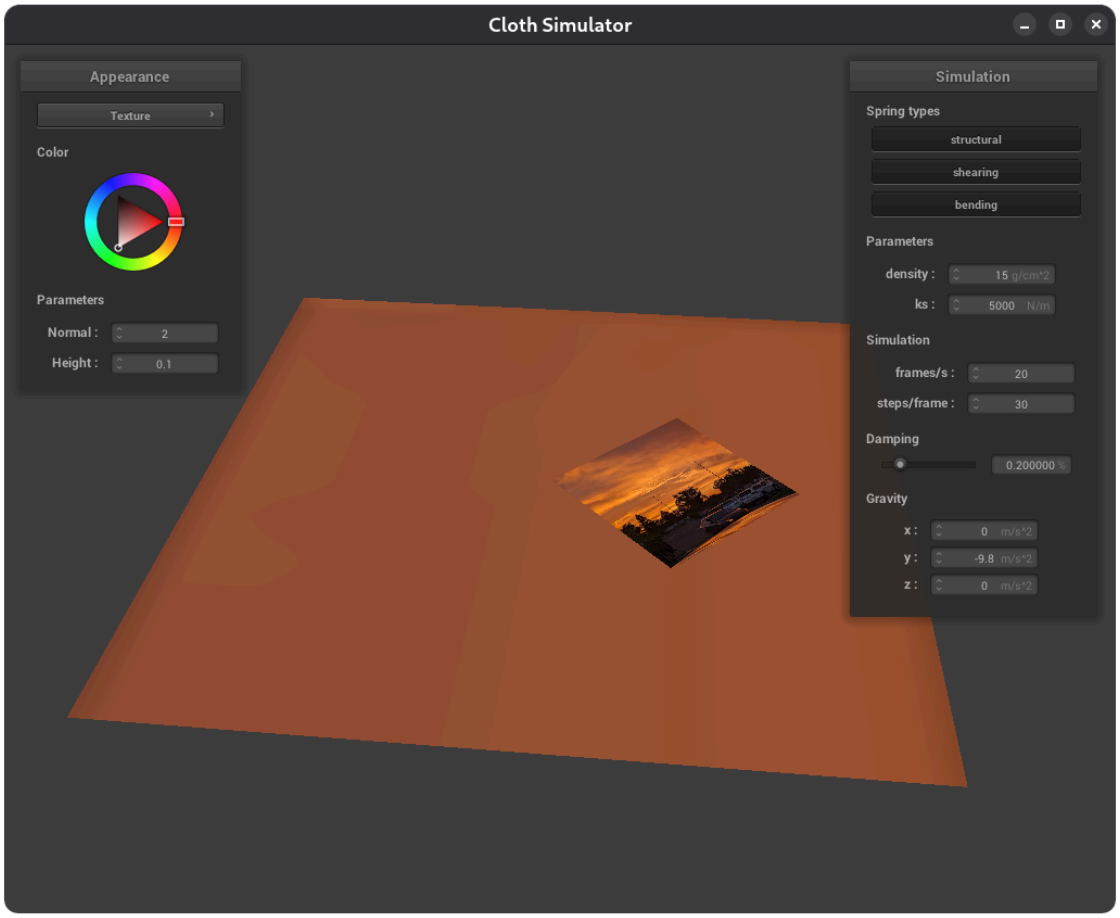


`ks=5000`



`ks=50000`

To implement plane collisions, we went back to ray tracing. We represent the change in position between the last position and current position as a ray. We find the time `t` that it takes to collide with the plane by solving the equation from the ray tracing homework. If this ray crosses the plane, we expect t to be between `0` and `1` (inclusive), since if `t < 0`, the collision happens before the last position, and if `t > 1`, the collision happens after the current position. If this condition is met, we calculate the correction vector based on multiplying the ray by the time it takes to reach the surface, then apply the correction vector as described, then adding the `SURFACE_OFFSET` multipled by the normal vector. If the vector is coming in from a direction opposing the normal vector, we flip the normal vector to allow this to still work.



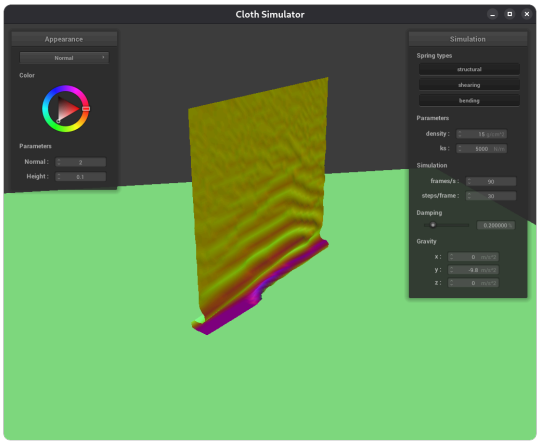
Cloth laying peacefully on the plane

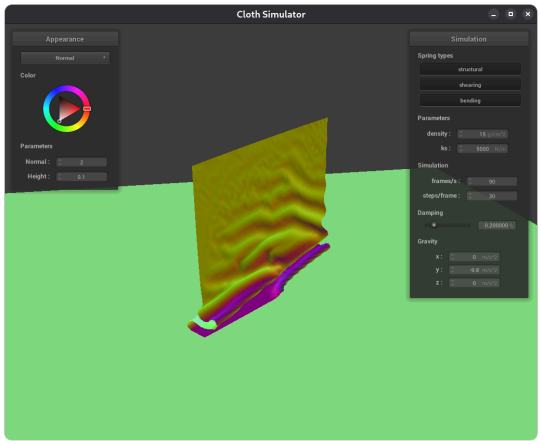
Part 4: Handling Self Collisions

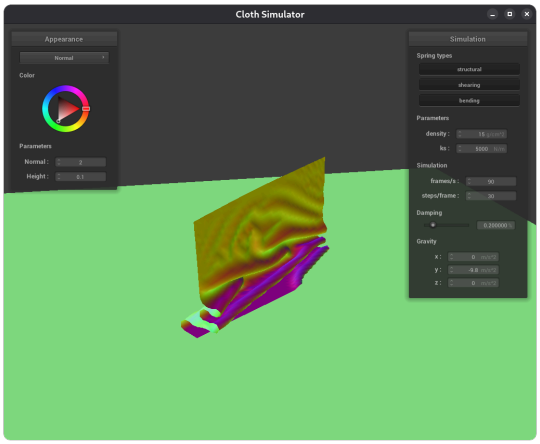
To implement self-collisions, we start by implementing a hash function, `hash_position`. This hash function takes in a set of x,y,z coordinates. Given these coordinates, it finds which of a set of fixed-size volumes that the box resides in, and returns the index of that box in row-major order. This gives an index for each position so that we can construct a hash map based on spatial locality. From there, we build a spatial map by using the hash position to index `this->map`, and storing in each entry a vector of pointers to point masses within that box. We reconstruct our spatial map on every iteration of the simulation. To handle the collision itself, we iterate over every point being simulated, get and get the box which the point is in. For each eligible point in the same box, we apply a correction vector to the original point by finding the distance between the two points and adding to an ideal correction vector the unit distance multipled by twice thickness minus the norm of the distance. In order for a pair of points to be eligible, they cannot be the same point, and the norm of the distance between them must be less than `2 * thickness`. Finally, to apply our correction to a point, we take the correction vector that we have been adding to, divide by

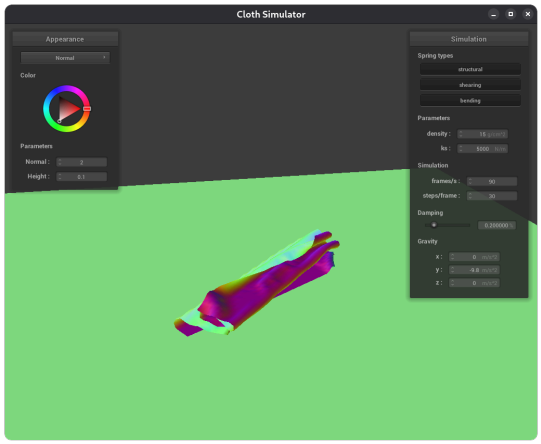
the number of corrections we have added to it, and then divide by the `simulation_steps` before adding this vector to the point's position.

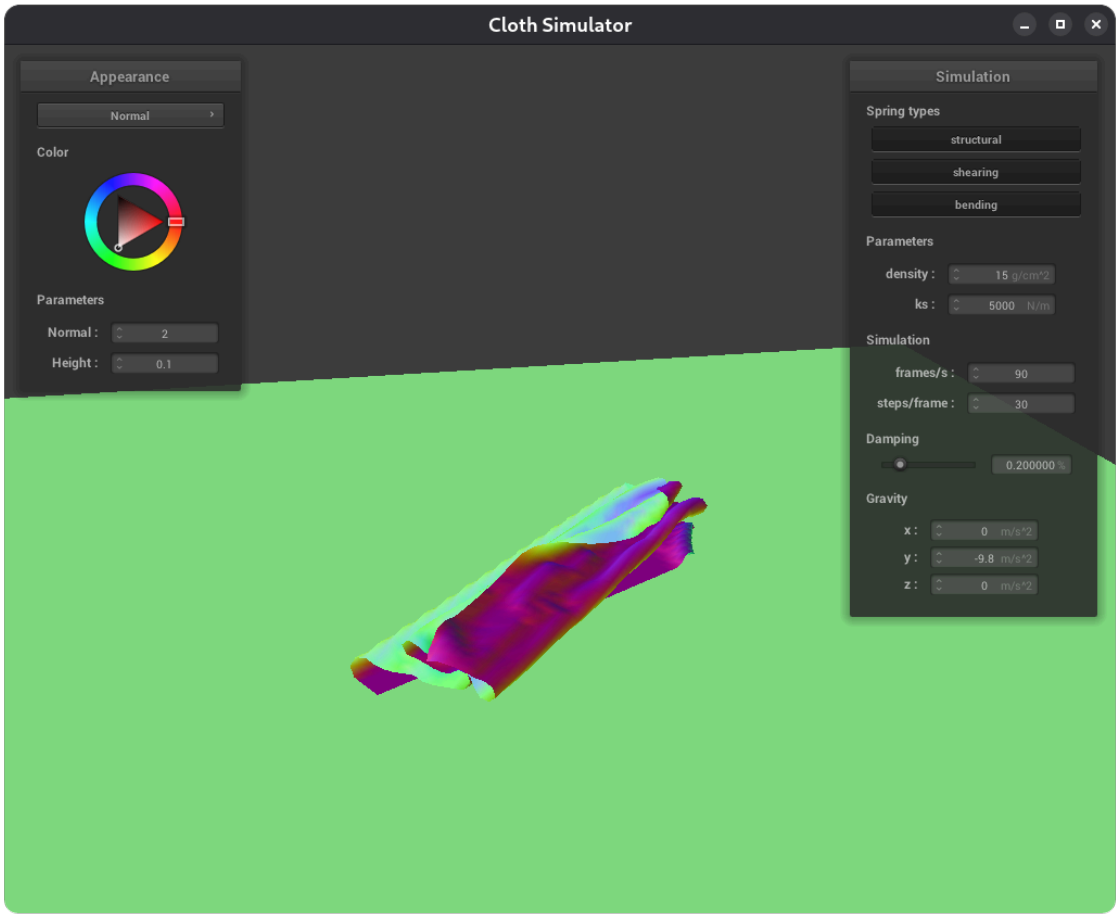
The result is a cloth which slowly falls onto itself:



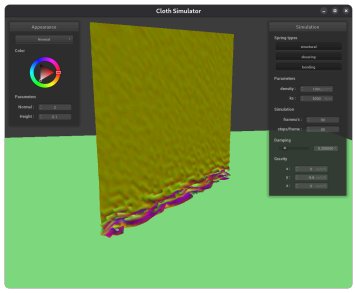


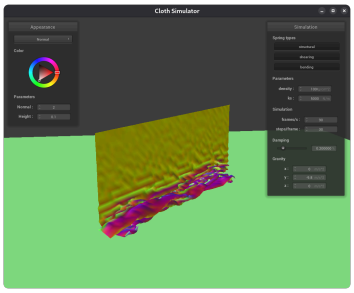


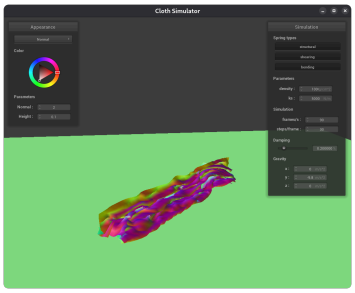




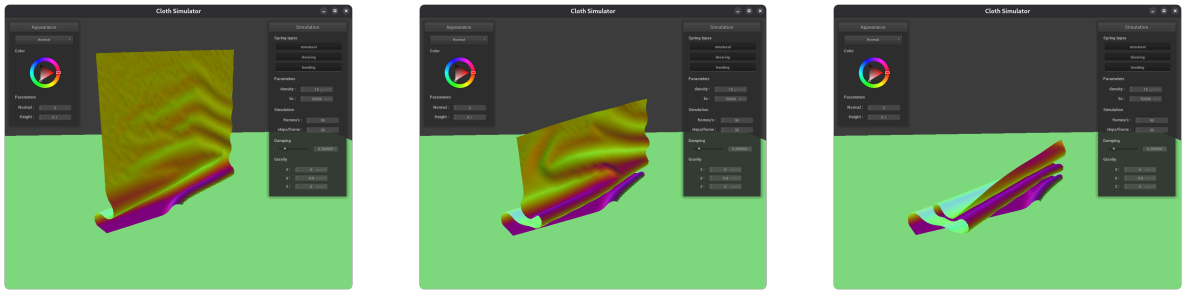
When we increase the density, we see that the cloth is less willing to spread out on the table, resulting in a considerably larger number of wrinkles and folds:







When we increase the ks, we see that the cloth tends to wrinkle less, resulting in a more spread out cloth with fewer folds:



When we decrease the density, we get similar effects to increasing the ks, and when we decrease the ks, we get similar results to increasing the density. Thus, we can conclude that the foldiness of the cloth is directly correlated to the density and inversely correlated with ks.

Part 5: Cloth Sim

A GLSL shader program is split into two parts – a vertex shader and a fragment shader. The vertex shader defines any movements to the actual geometry necessary to implement the shader (such as moving our points along the direction of the height in the displacement shader, or making any other changes to verticies or vertex normals). This information is then passed to a fragment shader, which uses the geometry produced by the vertex shader and samples points along it to produce the specific color that the shader displays at that point. Much like in Homework 1, this sampling is not necessarily per-pixel, it occurs per a variable arbitrary unit, which might result in supersampling or subsampling depending on what the relationship between that unit and the number of pixels on the display is.

Blinn-Phong

The Blinn-Phong shading model is a way to implement realistic local shading without having to do full global illumination. Blinn-Phong essentially breaks down into three separate shaders which are each multiplied by a set coefficient and layered on top of each other to get a desired effect.

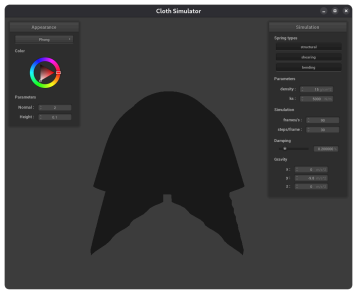
The simplest component of Blinn-Phong is an ambient shader, which represents passive light that is not dependent on the position of a fragment or the viewing angle. A real-world analogue of this is the way that the general brightness of a room impacts an object. This is just computed as the color and brightness of the light multiplied by the ambient coefficient.

The next component is a diffuse shader. This represents position-dependent light. A real-world analogue is the result of a nearby light casting onto our object, such that points closer to the light are brighter, and points occluded by the rest of the object are unlit as a result of the light. Implementing this results in multiplying three components. The first is the intensity of the light divided by the square of the radius to the light, which represents the diminishing brightness of a point light as you get further away from it. The second is the dot product between the normal vector at the point and the vector pointing from the point to the light (with a minimum value of

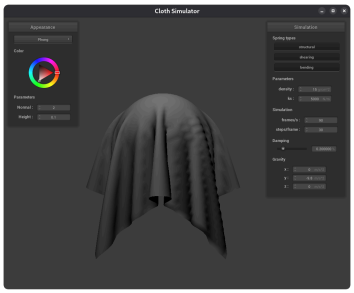
0). This represents the degree of occlusion of the light (IE: if the two vectors are exactly the same, the light is directly exposed to the point. If they are orthogonal, the light is not at all exposed to the point, and if the angle is obtuse, the result is a negative number which is bounded to 0 and treated as fully occluded). Finally, these two components are multiplied by a diffuse coefficient.

The final component is a specular shader. This represents light which is dependent on the view angle, such as the glare that bounces off of a shiny object. Implementing this is also breaks down into three components. The first component is just the same light intensity divided by square radius from the diffuse component, done for exactly the same reason. To find the second component, we first have to find the *bisector* between the view angle and the light angle, which is just the unit sum of the two vectors. This bisector represents the vector along which the reflection is occurring. Then, we take the dot product of the normal vector and this bisector to figure out how much of the reflection actually maps onto the surface (with a minimum of 0). Finally, we raise this dot product to an aribtrary power, **p**. Since the result of the dot product is the same thing as the result of taking the *cosine* of the angle between the normal and bisector, incrementing **p** results in a smaller resultant reflection. At the same time, though, values near 1 will be impacted slower than values near 0 by an increasing value of **p**, which allows us to vary *how reflective* a given object is by increasing or decreasing **p**. The final component is the specular coefficient.

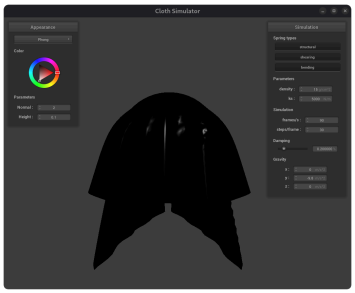
Finally, summing these three components together gives us the Blinn-Phong model of local shading.



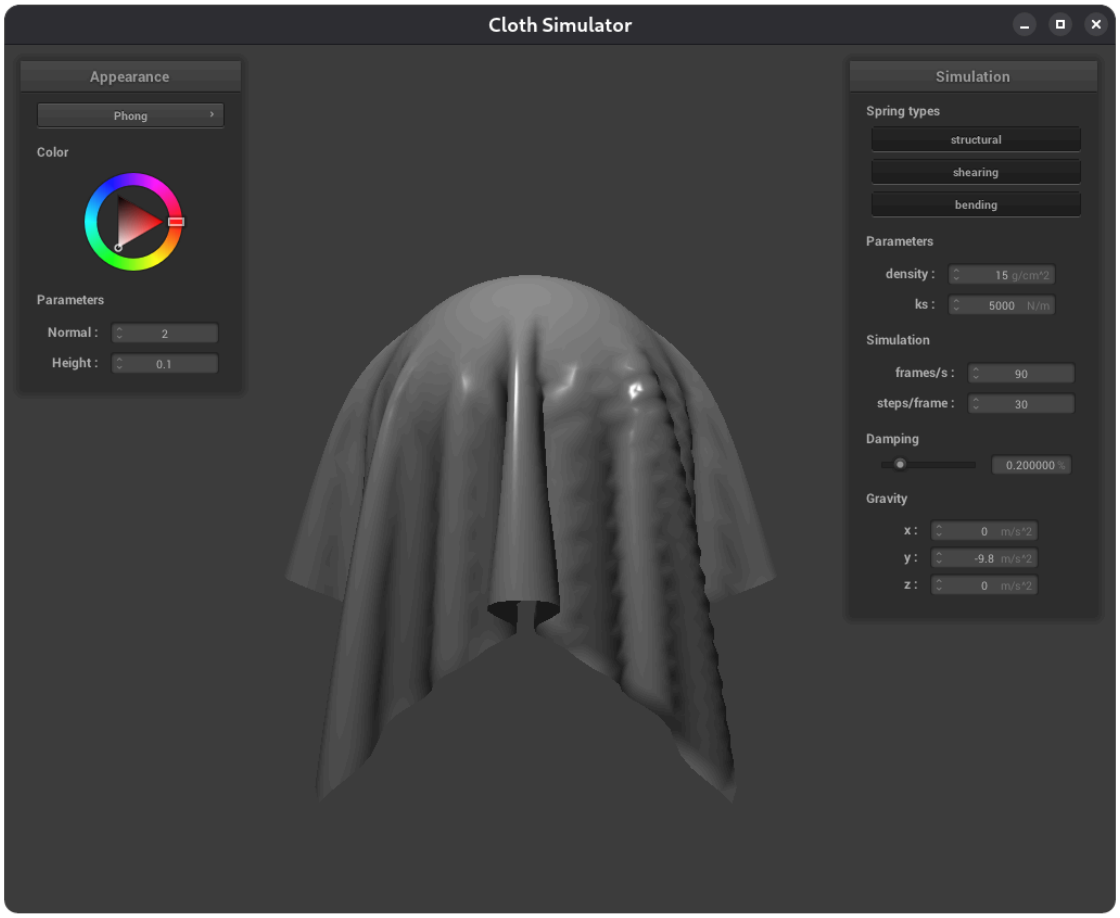
$k_a = 0.1$, $k_d = 0$, $k_s = 0$



$k_a = 0$, $k_d = 0.7$, $k_s = 0$



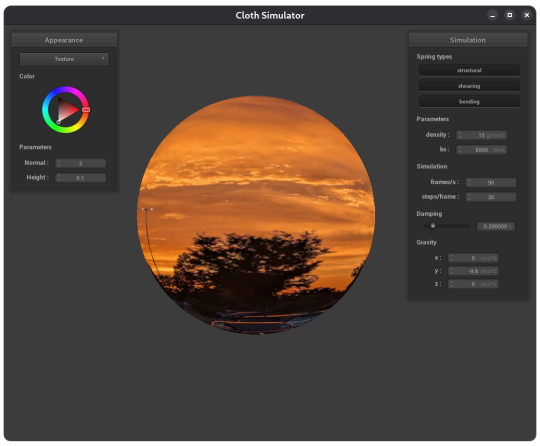
$k_a = 0$, $k_d = 0$, $k_s = 1$



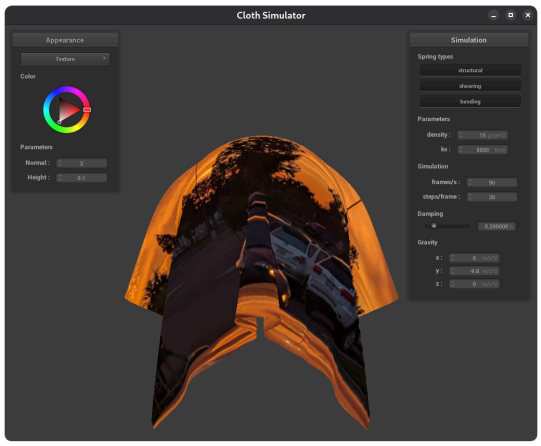
Blinn-Phong Shading with all three components enabled.

Texture Mapping

Texture mapping simply works by sampling the texture at the given uv-coordinate, and assigning that to the output color.



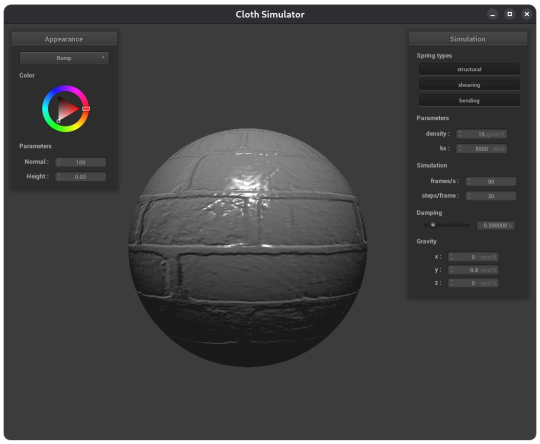
A Fresno sunset mapped onto the ball



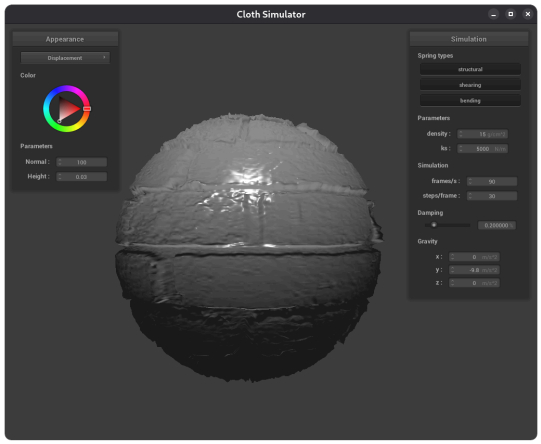
A Fresno sunset mapped onto the cloth

Bump and Displacement Mapping

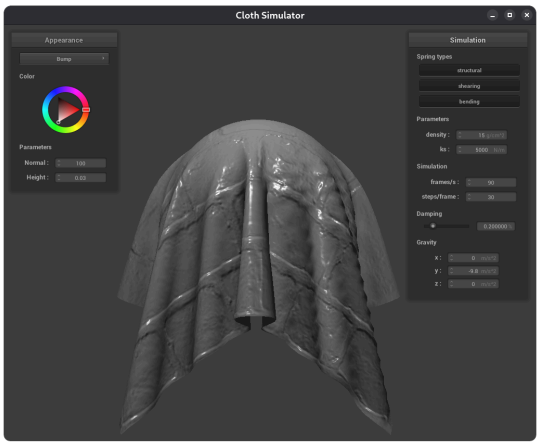
Bump mapping is a technique by which we alter the normals based on a specific image in order to give the impression that a surface has a specific texture. With displacement mapping, we implement bump mapping but we also actually modify the geometry, moving pixels based on the texture as well. We can see the result of these on the sphere here:



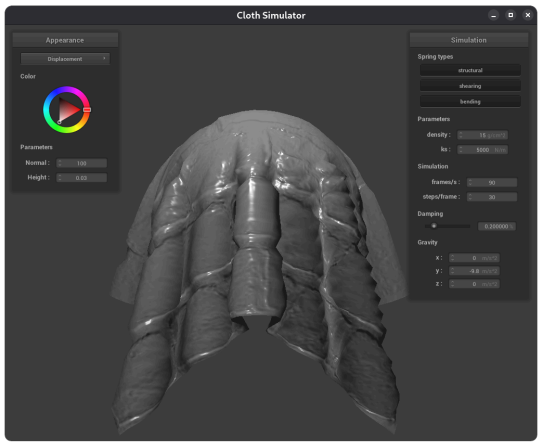
Sphere with Bump Shader



Sphere with Diffuse Shader

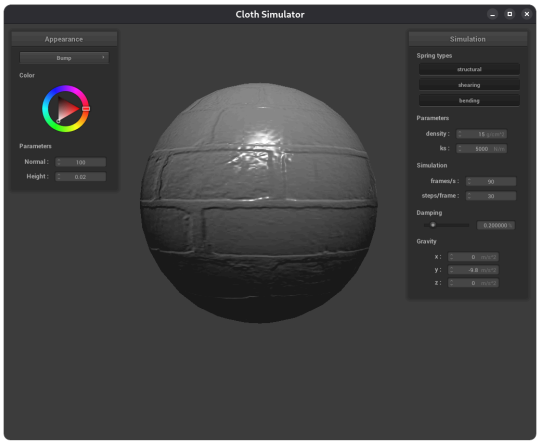


Cloth with Bump Shader

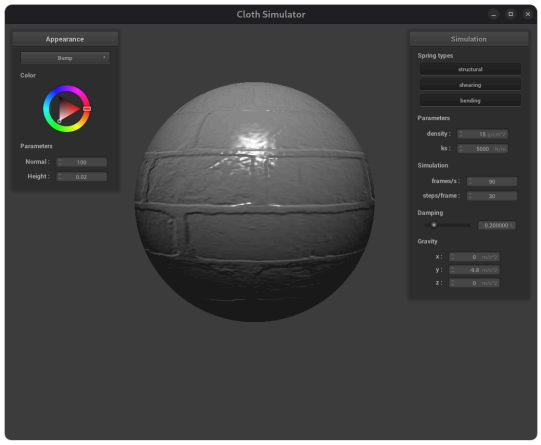


Cloth with Diffuse Shader

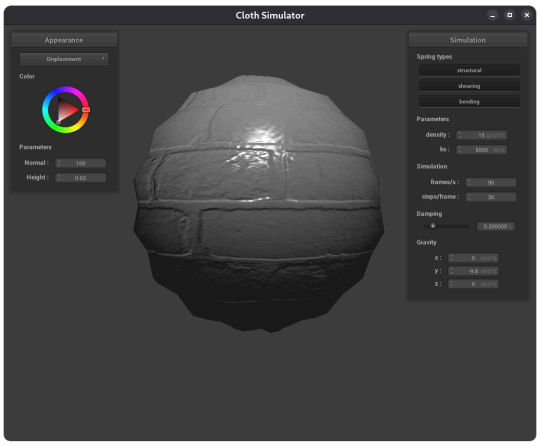
Some examples of bump and displacement mapping with low and high resolution meshes are shown below:



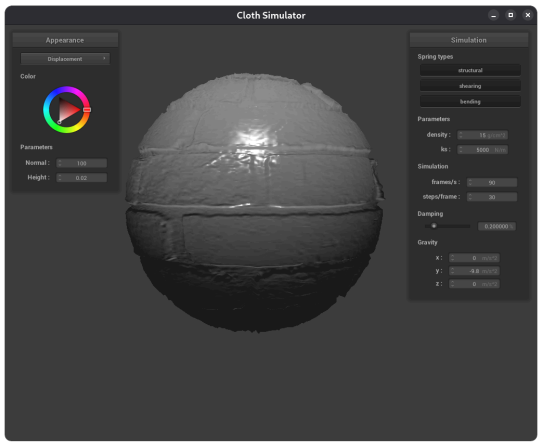
Bump shading with `-o 16 -a 16`



Bump shading with `-o 128 -a 128`



Displacement shading with `-o 16 -a 16`

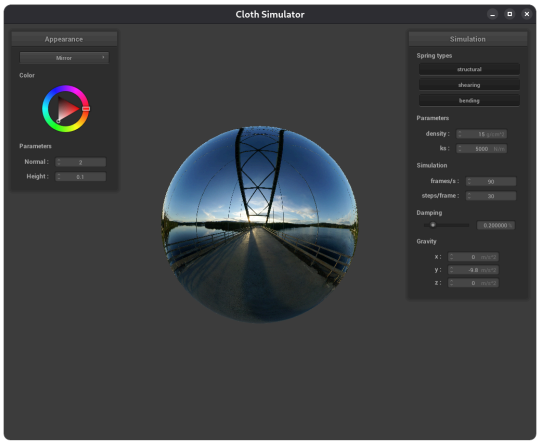


Displacement shading with `-o 128 -a 128`

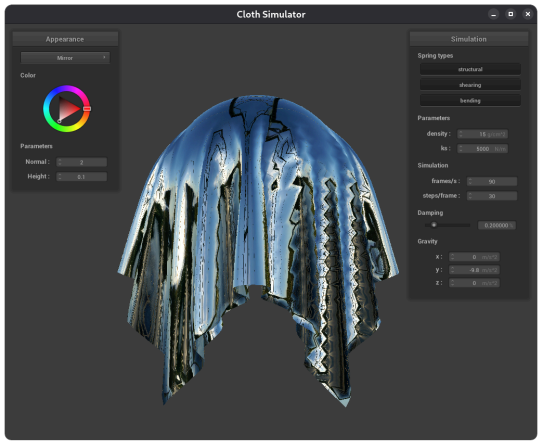
The bump shaders basically look the same across both, but the displacement shader is applied to verticies, so the model with more verticies looks more correct. The model with less verticies have a couple of jagged verticies outwards, but the displacement shader doesn't really reflect the model, since there aren't enough verticies to do that. The model with more verticies has a height map that accurately resembles the image itself.

Environment-Mapped Reflections

With environment-mapped reflections (also known as a mirror shader), we essentialy surround the scene with a box containing our "environment." To sample from this box, we trace rays from the surface to the camera, then reflect them across the normal and sample that resulting spot on the box.



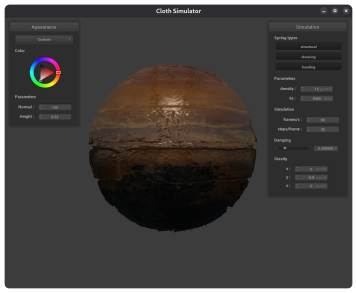
Mirror shading on the ball



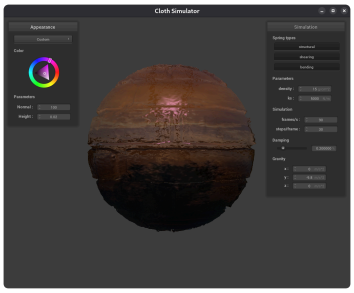
Mirror shading on the cloth

Custom Shader (Extra Credit)

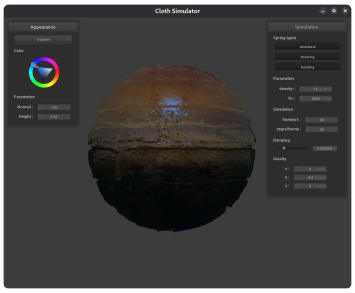
For the custom shader, we wanted to create a textured, colored glossy material with some amount of reflectivity, like many mugs or ceramic figures. We also wanted support for colored lighting. To do this, we began with our displacement shader. The first thing we added was the mirror effect, by copying in the mirror code (making sure to adjust the normals to be using the ones generated by the bump shader) and adding a new coefficient k_m to determine how reflective it should be. Next, to color the material we grabbed the texture from texture_1, and multiplied both the diffuse and ambient components by this value per-pixel. Finally, to add the colored lighting, we multiply just the ambient and spectral components by the normalized light color. The result is pictured below:



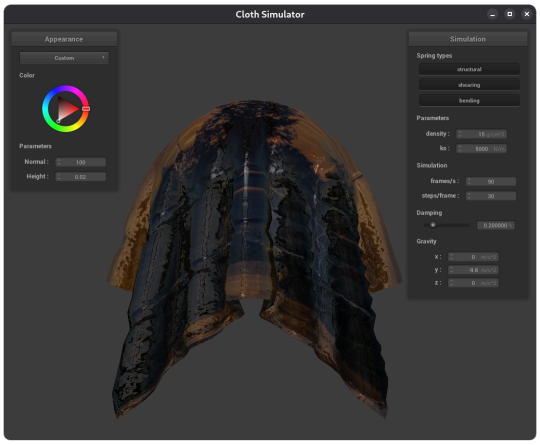
Sphere w/ custom shader, white light



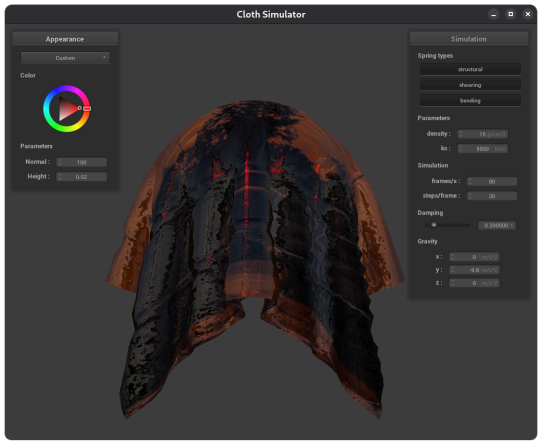
Sphere w/ custom shader, pink light



Sphere w/ custom shader, blue light



Cloth w/ custom shader, white light



Cloth w/ custom shader, red light