CS 284A: Computer Graphics and Imaging, Spring 2019

Project 4: Cloth Simulator

George Zhang, CS184-georgebzhang

Overview

NOTE: To easily view the videos, please view my final report on: https://georgebzhang.github.io/projects/Project_4/index.html.

In this project, I implemented a PointMass and Spring model to model the physics of a cloth. Using numerical simulation, I calculated the positions of each point of the cloth as a function of time by accounting for gravity, spring, and constraining forces. I learned how to handle collisions of the cloth with other objects using geometric relations as well as with itself using spatial hashing for an efficient approximation. Finally, I explored GLSL shader programs, isolated programs that run in parallel on the GPU rather, resulting in much faster rendering times. This is in contrast with previous projects, which involved raytracing computation on the CPU and long rendering times.

Part 1: Masses and springs

Process:

1: Calculate differential width and height: d_width = width / (num_width_points - 1), d_height = height / (num_height_points - 1)

2: If orientation is HORIZONTAL, emplace_back PointMasses with parameters Vector3D(x, 1, z), false into point_masses, where x and z encompass the full width and height. x and z are incremented by d_width and d_height, respectively. Using false here assumes that all PointMasses are not pinned. This will be corrected in a later step.

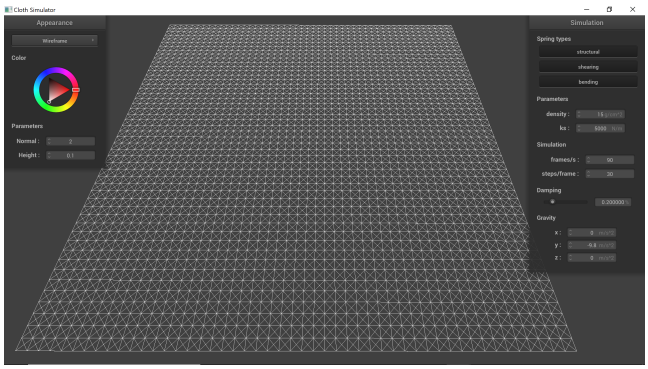
3: If orientation is VERTICAL, emplace_back PointMasses with parameters Vector3D(x, y, z), false into point_masses, where x and y encompass the full width and height. x and y are incremented by d_width and d_height, respectively. z is a small random offset ranging from -1/1000 to 1/1000. Using false here assumes that all PointMasses are not pinned. This will be corrected in a later step.

4: For each pinned PointMass, update its pinned to true.

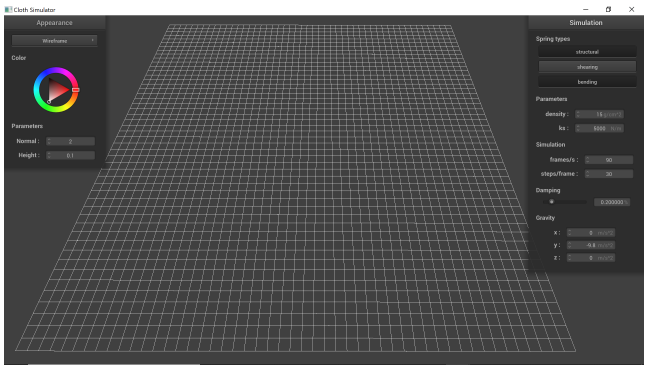
5: emplace_back Springs between each pair of PointMasses that satisfy the structural constraint.

6: emplace_back Springs between each pair of PointMasses that satisfy the shearing constraint.

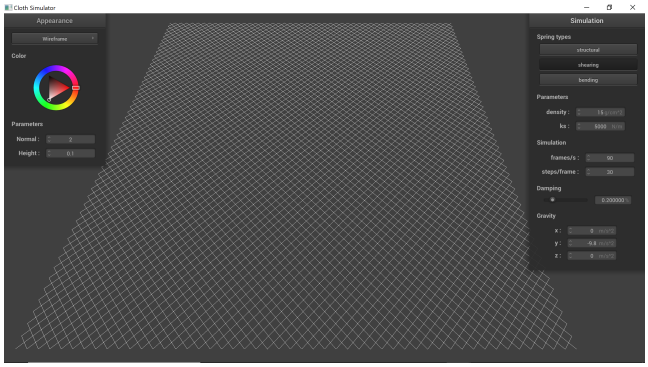
7: emplace_back Springs between each pair of PointMasses that satisfy the bending constraint.



All constraints



Only structural constraints



Only shearing constraints

Structural constraints exist between a point mass and the point mass to its left as well as the point mass above it.

Shearing constraints exist between a point mass and the point mass to its diagonal upper left as well as the point mass to its diagonal upper right.

Part 2: Simulation via numerical integration

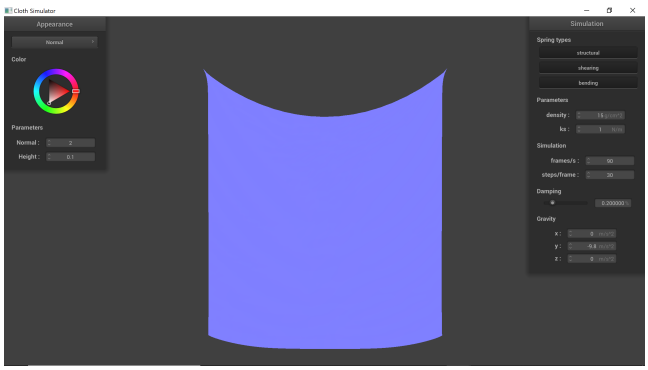
Process: For each call to Cloth::simulate:

1: For each PointMass, reset its forces to mass * external_accelerations[0]

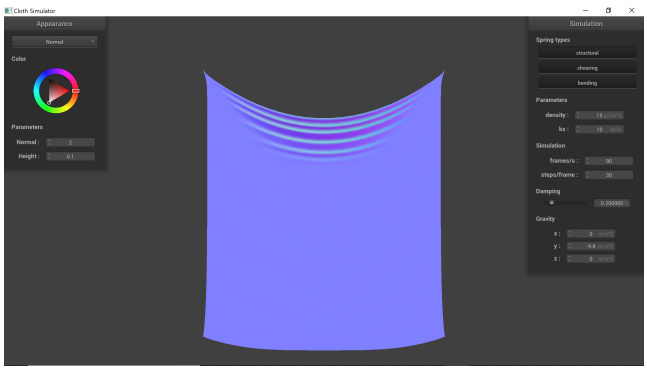
2: For each Spring, only if it has some constraint enabled, calculate $Fs = ks * (||pa - pb|| - l)$. If the spring type is BENDING, multiply Fs by 0.2. The spring force vector Fsv is a unit vector in the direction of PointMass a to PointMass b (pm_b->position - pm_a->position) scaled by Fs. This direction is important. In this case, we add Fsv to PointMass a's forces and subtract Fsv from PointMass b's forces.

3: For each PointMass, calculate $x_{t+dt} = x_t + (1 - d) * (x_t - x_{t-dt}) + a_t * dt^2$. Set its last_position to its current position.

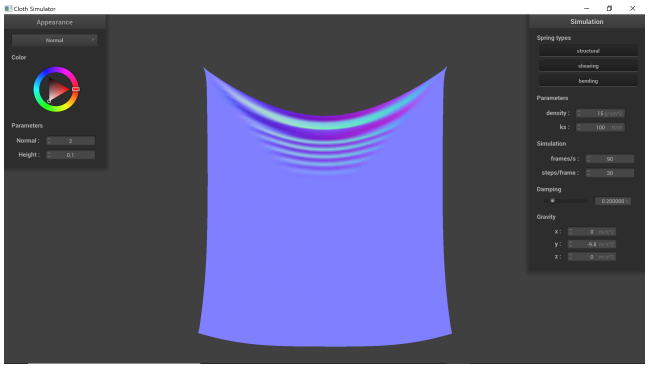
4: For each Spring, only if not both PointMasses are pinned and the distance between both PointMasses is less than 110% of its rest length, calculate the difference $diff = ||pa - pb|| - 1.1 * restlength$. The difference vector diffv is a unit vector in the direction of PointMass a to PointMass b scaled by diff. If either PointMass is pinned, apply this full diffv to the other PointMass, otherwise apply half this diffv to each PointMass. Keep direction in mind!



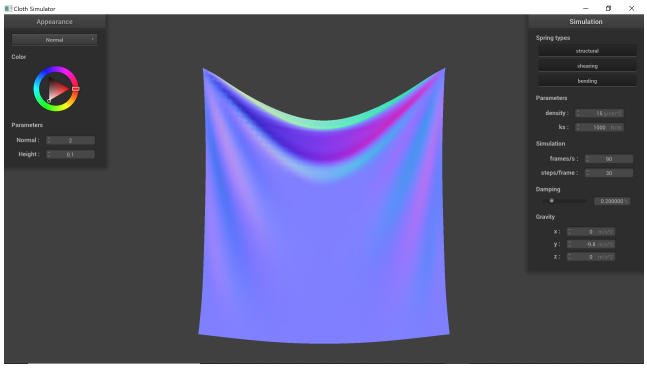
k = 1



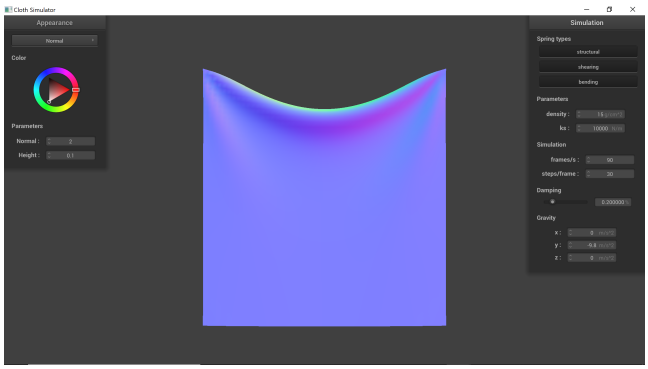
k = 10



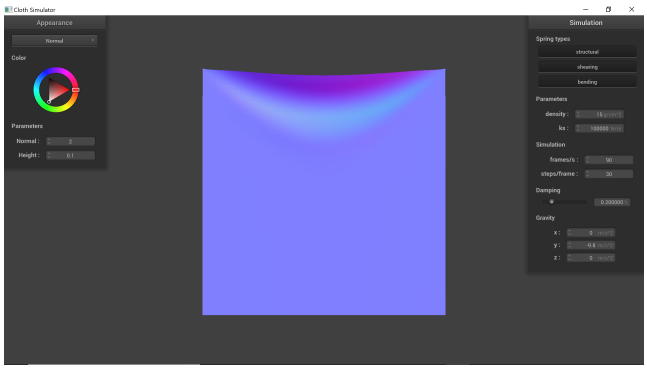
k = 100



k = 1000

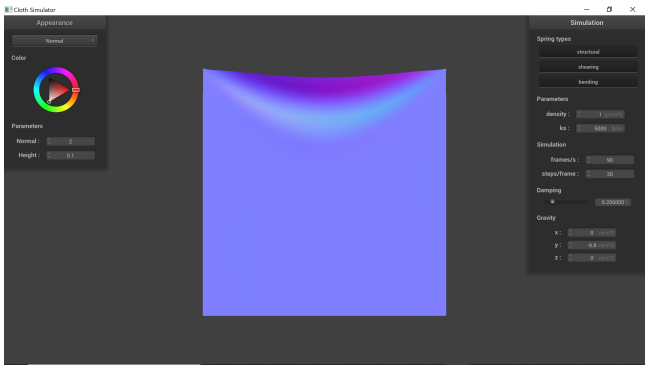


k = 10000

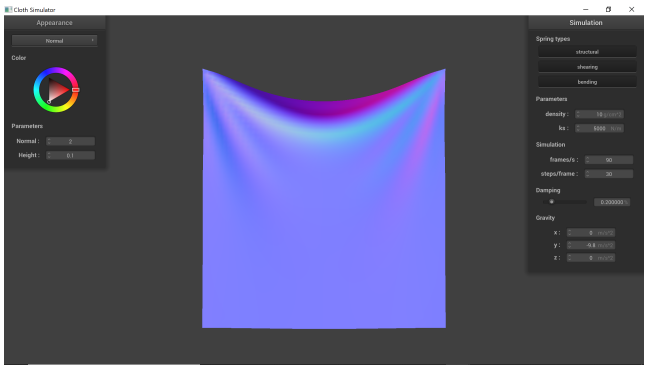


k = 100000

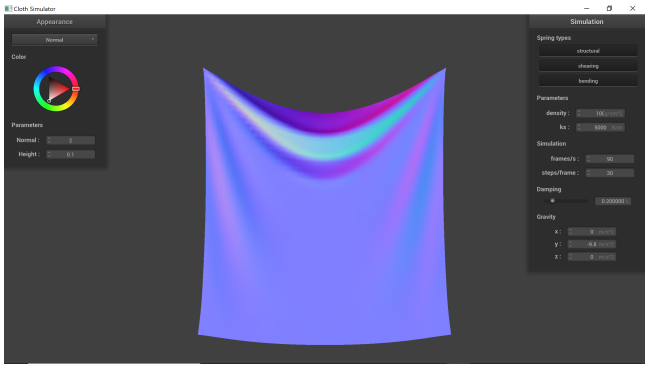
In general, as k of a spring increases, its stiffness increases. From the images above, note that as k increases, the curvature originating from the two pinned PointMasses decreases. Also, the cloth does not droop as low with higher k. These observations align with the physical understanding that the stiffness of a spring increases as its k increases. Note that in the very low k (k = 1) and the very high k (k = 100000) cases, there are few wrinkles in the cloth. For very low k, there is almost no sense of stiffness (the cloth is almost perfectly loose), thus we should expect few wrinkles. For very high k, the cloth is almost perfectly stiff, thus we should also expect few wrinkles. The maximum number of wrinkles should occur for medium values of k, where the cloth is between perfectly loose and perfectly stiff.



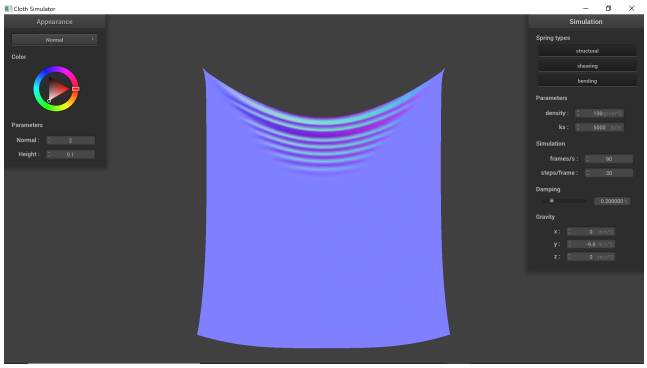
density = 1 g/cm^2



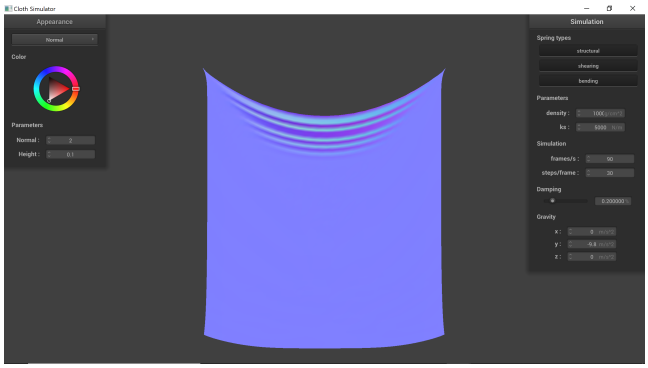
density = 10 g/cm^2



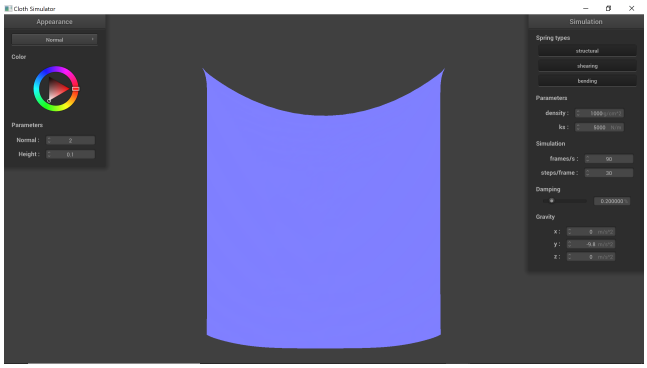
density = 100 g/cm^2



density = 1000 g/cm^2

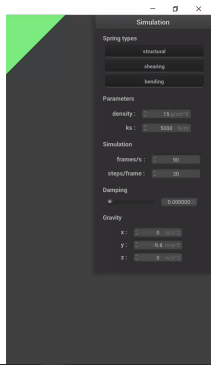


density = 10000 g/cm^2



density = 100000 g/cm^2

From the images above, note that as density increases, the curvature originating from the two pinned PointMasses increases. Also, the cloth does not droop as low with lower density. Note that in the very low density (1 g/cm^2) and the very high density (100000 g/cm^2) cases, there are few wrinkles in the cloth. The maximum number of wrinkles seems to occur for medium values of density.

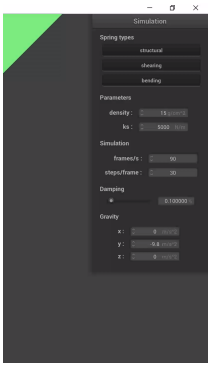


0:00 / 1:25

damping = 0%

0:00 / 1:10

damping = 0.05%

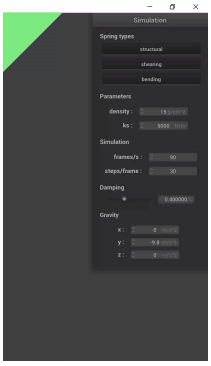


0:00 / 0:30

damping = 0.1%

0:00 / 0:14

damping = 0.2%

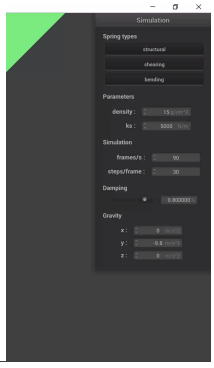


0:00 / 0:34

damping = 0.4%

0:00 / 1:02

damping = 0.6%



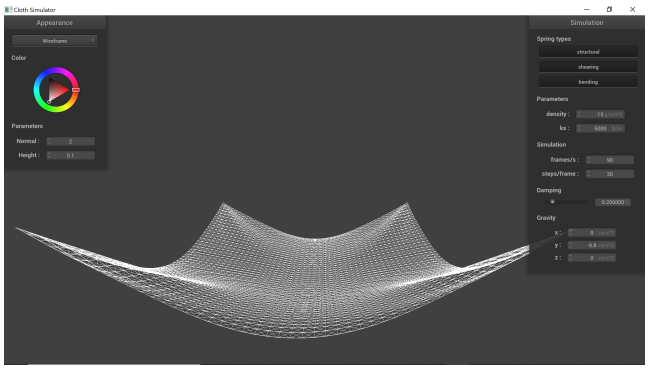
0:00 / 1:10

damping = 0.8%

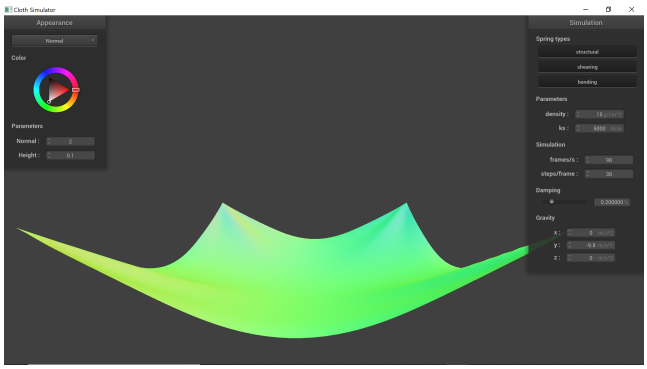
0:00 / 1:05

damping = 1%

In general, as damping (friction, heat loss, etc...) of a system increases, the faster energy is lost in the system. This is reminiscent of a pendulum; if there is little friction at the point of suspension and little air resistance (perhaps the system is in vacuum), then the pendulum will swing for longer than it will in an environment where there is more friction and resistance. From the videos above, note that as damping increases, the less time it takes for the cloth to reach its rest state. These observations align with the physical understanding that higher damping results in faster energy loss.



pinned4.json wireframe



pinned4.json normal

Shown above are the wireframe and normal images of a cloth pinned at 4 corners.

Part 3: Handling collisions with other objects

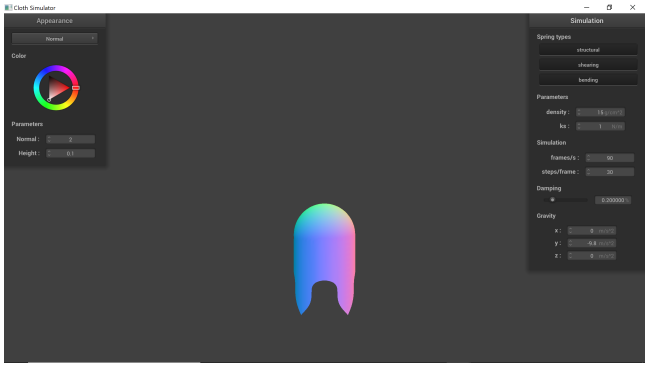
Process (Cloth-Sphere Intersection):

1: Calculate the vector connecting the sphere's origin and the PointMass' position. If the distance of this vector is less than the sphere's radius, then the PointMass is inside the sphere. In this case, we proceed with the following steps:

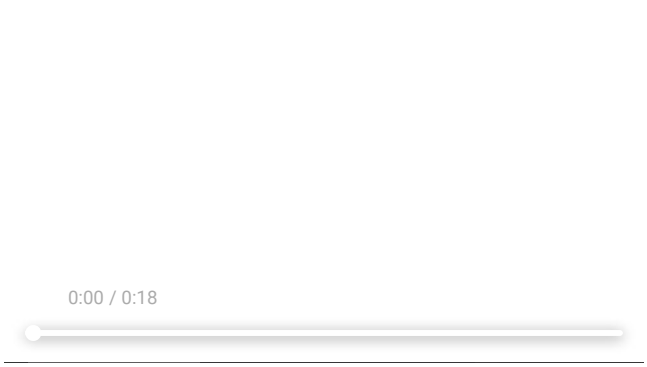
2: The tangent point is calculated by scaling a unit vector in the direction from the sphere's origin to the PointMass' position by the sphere's radius.

3: The correction vector is calculated by subtracting the PointMass' last_position from the tangent position calculated above.

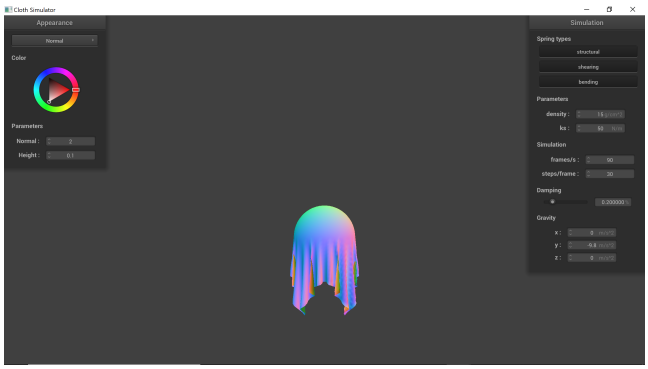
4: The PointMass' position is set to its last_position plus (1 - friction)*correction vector.



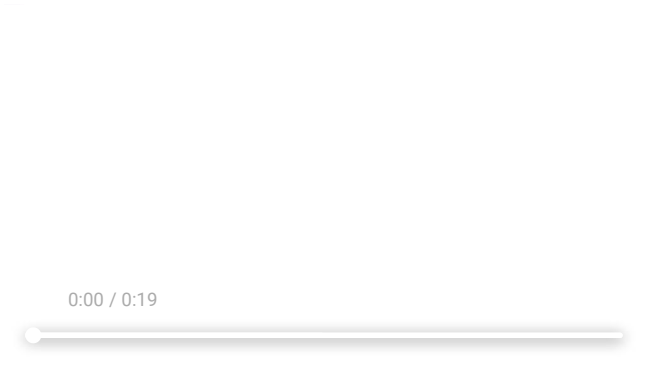
k = 1



k = 1



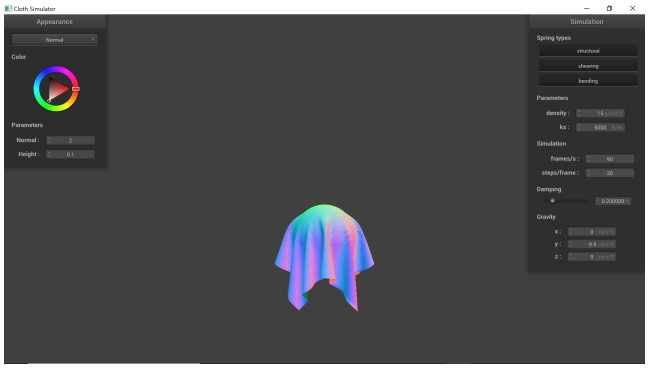
k = 50



k = 50

k = 500

k = 500

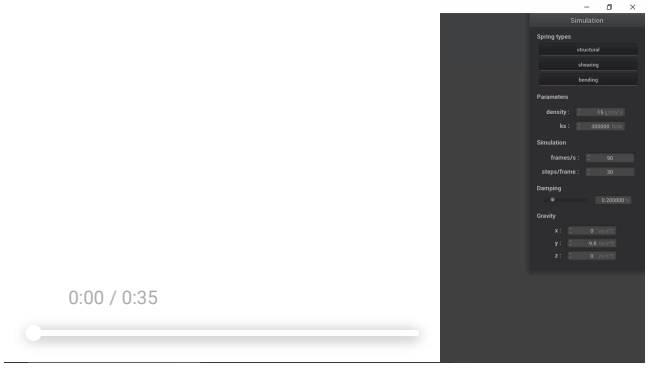


k = 5000

k = 5000

k = 50000

k = 50000



k = 300000

k = 400000

As discussed previously, as k of a spring increases, so does its stiffness. Thus, when k is low (k = 1), the cloth stretches and droops lower, since its springs have very low stiffness. When, k is high (k = 50000), the cloth does not droop as much, as its springs are much stiffer. Increasing k even more (k > 300000), the cloth becomes so stiff that it bends very little and easily falls off the sphere.

Process (Cloth-Plane Intersection):

1: Given a PointMass, we can build a ray with origin as PointMass' last_position and direction as the plane's normal vector. However, we should build two rays with the same origin but opposite directions. This is because the PointMass' last_position may be on the same side of the plane as its normal or on the opposite side. Thus, if we only consider one ray, we might miss the plane.

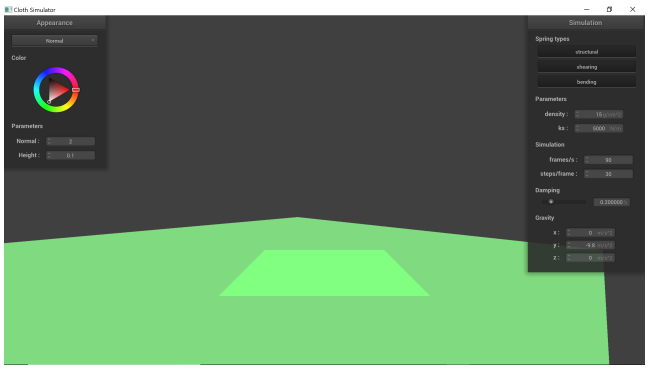
2: Solve for the time needed for the ray (starting from its origin) to intersect the plane, i.e. t = (p' - o).n / d.n, where . is the dot product, p' is a point on the plane, o is the origin of the ray, i.e. PointMass' position, and n is the normal of the plane. Since we have the two

rays from above, we calculate two times t1 and t2. Only if one of the times t >= 0 and infinity > t should we proceed with the following steps:

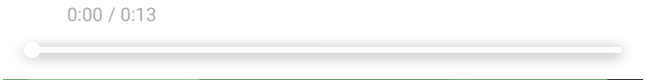
3: The tangent point is calculated as o + t * d, where o is the origin of the ray (PointMass' position), t is the time calculated above, and d is the direction of the ray (-normal).

4: The correction vector is initially calculated as tangent point - last_position (of the PointMass). We have to apply SURFACE_OFFSET to the correction vector by adding normal*SURFACE_OFFSET to the correction vector according to which side of the plane the PointMass is initially located.

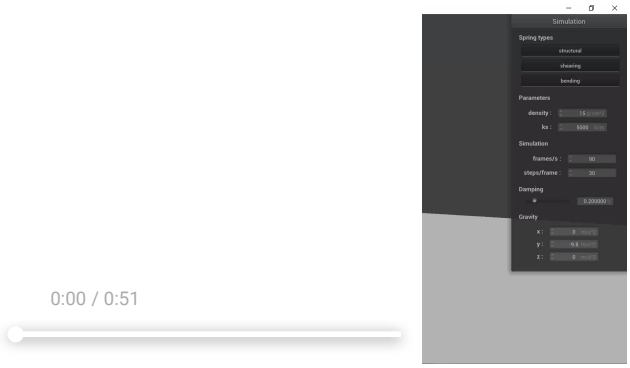
5: There is only a collision if the PointMass' last_position and position are on opposite sides of the plane. We can obtain two vectors tangent point - last_position and tangent point - position. If the dot product of the plane's normal with these two vectors are of different signs, then last_position and position are on opposite sides of the plane. Only in this case, PointMass' position is set to its last_position plus (1 - friction)*correction vector.



Cloth plane intersection



Cloth (horizontal) plane intersection



Cloth (vertical) plane intersection (wireframe)

Cloth (vertical) plane intersection (normal)

Shown above is an image of a horizontal cloth resting on a plane and a video of the process. Also shown is the wireframe and normal videos of a vertical cloth falling on a plane.

Part 4: Handling self-collisions

Background: We previously implemented collisions between a cloth and another object. How do we implement collisions between a cloth and itself? The naive brute force approach is a double for loop among all PointMasses in the cloth. If two PointMasses are too close, then push them farther apart from each other. This is an O(n^2) solution. We can do better using spatial hashing. Imagine dividing the cloth into 3D boxes with their own sets of PointMasses. Somehow, we must hash the positions of PointMasses within the same 3D box into the same "bucket". We will rebuild this hash table for every call to Cloth::simulate(...). With this hash table, we will know which PointMasses are in close proximity to each PointMass. Thus, we no longer need a double for loop among all PointMasses. We just need one for loop among all PointMasses, then we can look up in the hash table which PointMasses are close to each PointMass. Note that this is an approximation. For PointMasses on the edges of these 3D boxes, we will not consider PointMasses next to it that are not within the same 3D box.

Process (Cloth::hash_position(...)):

1: We want to map/hash a PointMass' position to a 3D box. The cloth has total width and height of width and height (width = 1 and height = 1 for the cloth in selfCollision.json). The cloth is made up of num_width_points * num_height_points PointMasses. The space between PointMasses along the width is width/(num_width_points-1). The space between PointMasses along the height is height/(num_height_points-1). Assume that each 3D box is 3 of these width spaces wide and 3 of these height spaces tall. The third dimension of this 3D box is simply the larger of these previous two values.

2: From above, we have w = 3*width/(num_width_points-1), h = 3*height/(num_height_points-1), and t = max(w, h), where w, h, and t are the dimensions of a 3D box. A PointMass's position for the cloth in selfCollision.json (with VERTICAL orientation) ranges from 0 <=

pos.x <= 1, 0 <= pos.y <= 1, and -1/1000 <= pos.z <= 1/1000. What does trunc_x = pos.x / w represent? It represents the (trunc_x)th 3D box in the horizontal direction. Likewise, trunc_y = pos.y / h represents the (trunc_y)th 3D box in the vertical direction, and trunc_z = pos.z / t represents the (trunc_z)th 3D box in the z direction.

3: (trunc_x, trunc_y, trunc_z) represents a single, unique 3D box. We will use a simple prime rolling hash to convert (trunc_x, trunc_y, trunc_z) to a float, which will serve as a key for the hash table.

Process (Cloth::build_spatial_map(...)):

1: Clear the hash table map (map must be cleared for every call to Cloth::simulate(...)).

2: For each PointMass, compute its hash using Cloth::hash_position(...)). If the key does not exist in map, create a new vector and push_back the PointMass. Otherwise, just push_back the PointMass.

Process (Cloth::self_collide(PointMass &pm, double simulation_steps)):

1: Create a correct vector correctv = Vector3D(0, 0, 0). This will store the sum of all correction vectors.

2: Compute the hash of the PointMass pm using Cloth::hash_position(pm.position).

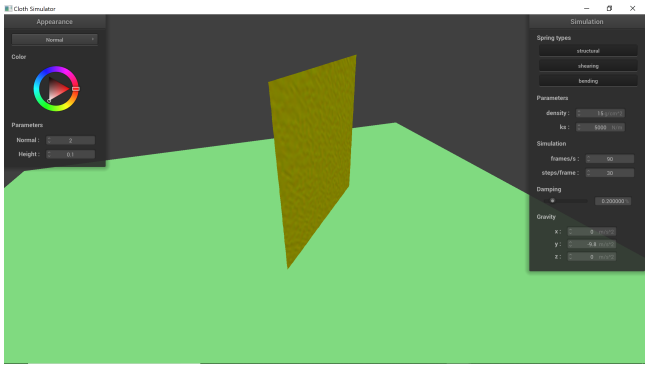
3: Create a count variable num = 0. For all the PointMass candidates pm_cand in the same 3D box as PointMass pm, only if pm_cand and pm are less than 2*thickness apart, proceed with the following step:

4: Add to correctv a vector in the direction of (pm.position - pm_cand->position) scaled by the difference between 2*thickness - and the distance between pm and pm_cand. Increment num (introduced above).

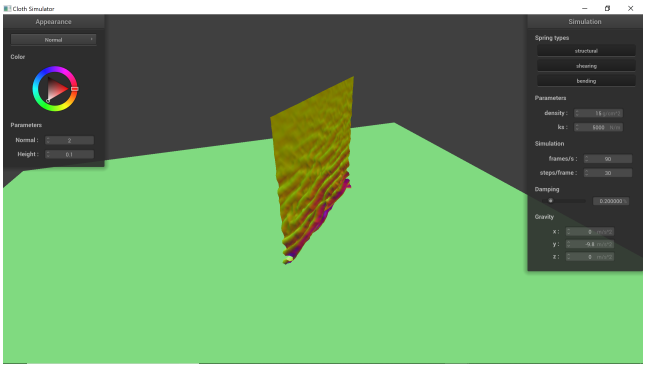
5: If num (number of PointMass candidates with distance to pm less than 2*thickness) is zero, return. Otherwise, divide correctv by num. Divide correctv again by simulation_steps. Add correctv to pm.position.

Process (Cloth::simulate(...)):

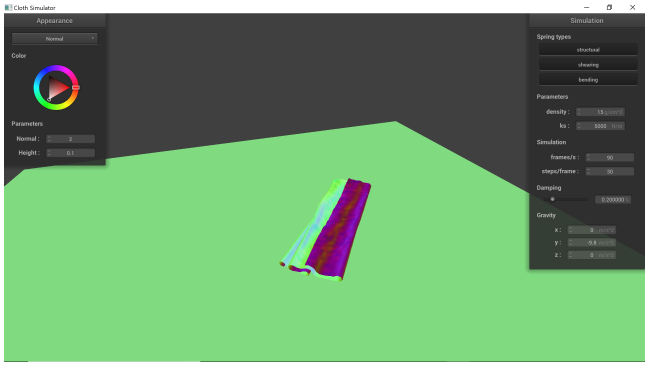
1: After the Verlet integration done in Part 2 and before the collision handling done in Part 3, call build_spatial_map(...). Then for each PointMass, call self_collide(...).



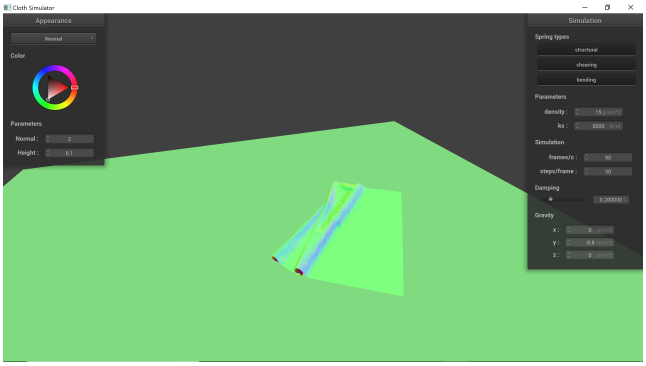
Snapshot 1



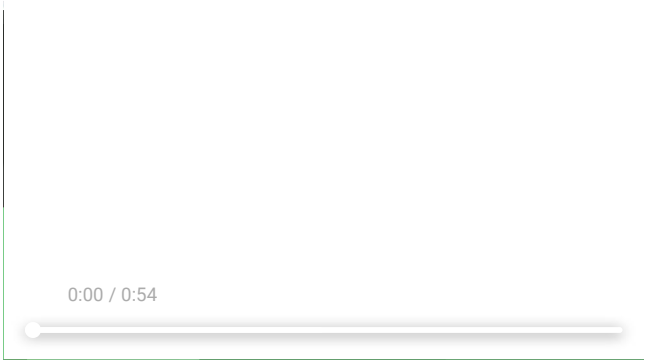
Snapshot 2



Snapshot 3

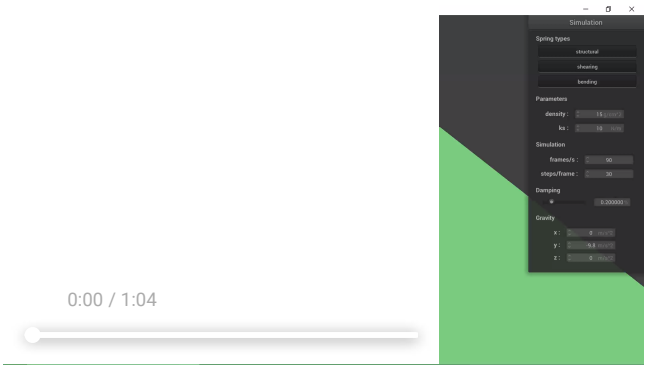


Snapshot 4

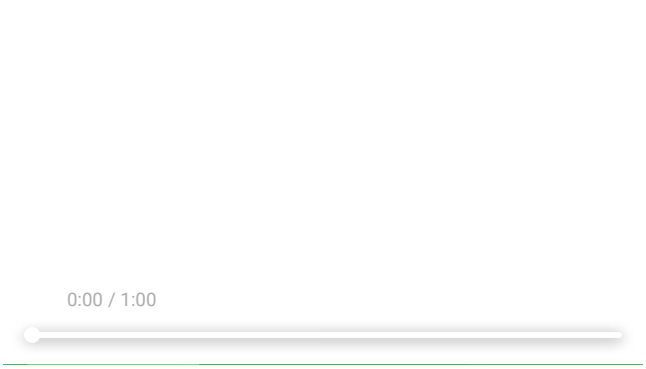


Self-colliding vertical cloth on a plane

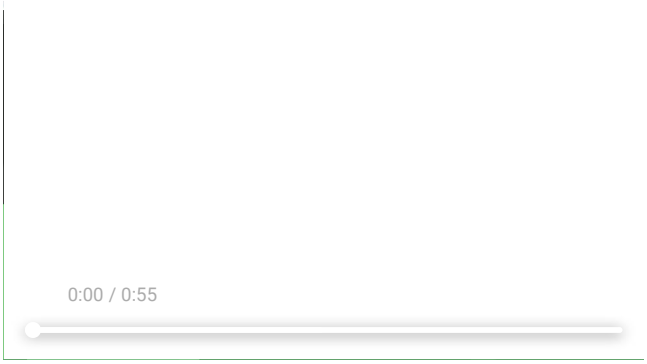
Above are snapshots of the self-colliding vertical cloth on a plane, as well as a video of the process.



k = 10

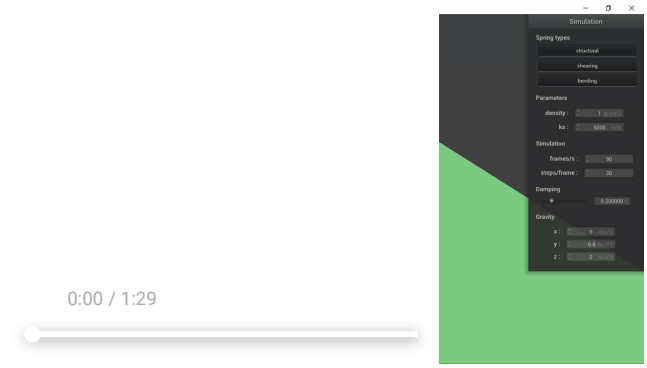


k = 1000

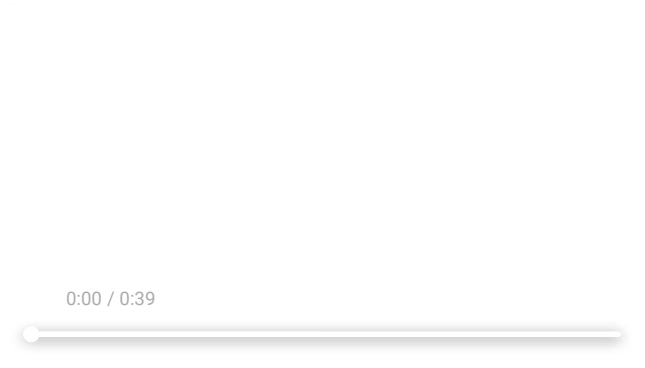


k = 100000

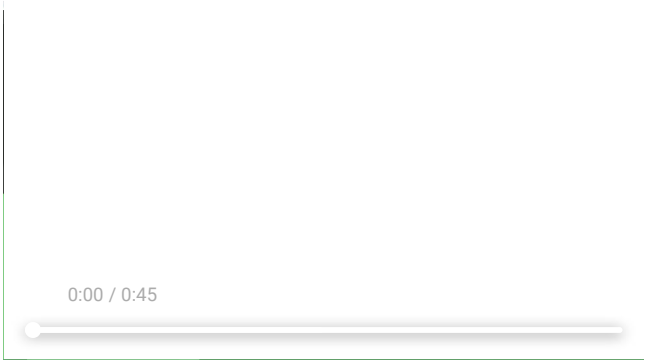
As k increases, so does the stiffness of the Springs connecting the PointMasses. Thus, as the videos shows, as k increases, the cloth tends to bend less, as the Springs are more stiff.



density = 1 g/cm^2



density = 10 g/cm^2



density = 100 g/cm^2

As shown in the videos, as density decreases, the cloth becomes more stiff. This is because with a lower density, there are less points at which the cloth can bend.

Part 5: Shaders

In my understanding, a mesh is divided into triangles (3 vertices each). The positions of these vertices are stored in a Vertex Array Object (VAO)'s attribute list (which we didn't have to do ourselves for this project). A vertex shader can access these positions as inputs (in vec3 in_position). The vertex shader may also have other inputs (which we can choose) and executes once for each vertex. gl_Position is assigned in the vertex shader, which determines where on the screen the vertex is rendered. Calculations (e.g. converting coordinates into other coordinate spaces, calculating color (of the current vertex), etc...) can also be done in the vertex shader, but note that these calculations only apply to the 3 vertices of the triangle (To be discussed later, the fragment shader interpolates these values for all pixels encompassed by these 3 vertices). Outputs of the vertex shader (e.g. out vec4 v_position) are calculated in the vertex shader and are inputs (e.g. in vec4 v_position) to the fragment shader. The vertex shader calculates values for the 3 vertices of the triangle, and the fragment shader calculates values for all the pixels inside the triangle (by interpolation, barycentric coordinates, etc...). Outputs of the fragment shader may include colors, and we can calculate these colors taking into account lighting.

Task 1: Diffuse Shading

$$\mathbf{L}_d = k_d(\mathbf{I}/r^2)max(0, \mathbf{n} \cdot \mathbf{l})$$

out_color = kd * intensity_falloff * max(0, dot(v_normal, normalize(l)));

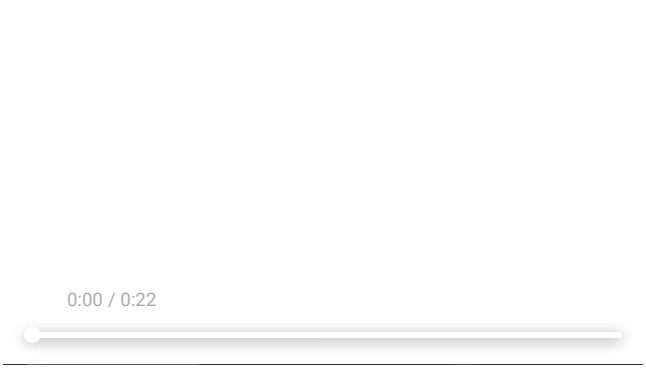
Diffuse shading

Diffuse shading

Shown above is an image and video of diffuse shading.

Task 2: Blinn-Phong Shading

Blinn-Phong shading

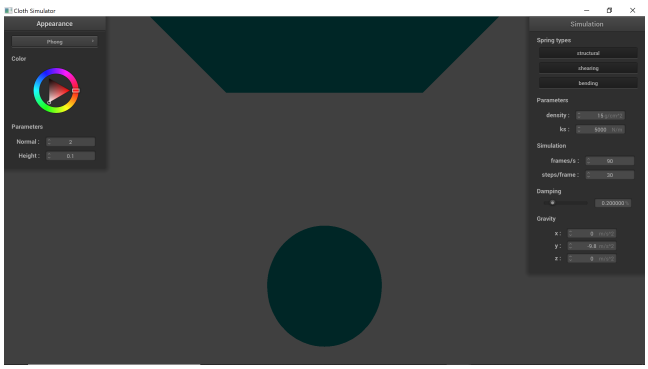


Blinn-Phong shading

The Blinn-Phong shading model includes (1) ambient shading, (2) diffuse shading, and (3) specular shading.

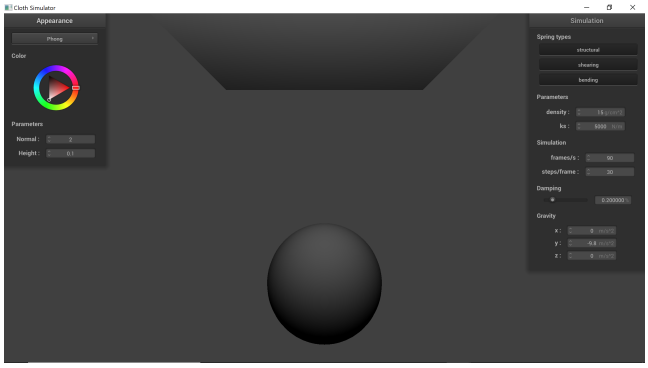
$$\mathbf{L}_{bp} = \mathbf{L}_a + \mathbf{L}_d + \mathbf{L}_s = k_a \mathbf{I}_a + k_d (\mathbf{I}/r^2) max(0, \mathbf{n} \cdot \mathbf{l}) + k_s (\mathbf{I}/r^2) max(0, \mathbf{n} \cdot \mathbf{h})^p$$

where $\mathbf{L}_{bp}$ is the Blinn-Phong intensity, $\mathbf{L}_a$ is the ambient intensity, $\mathbf{L}_d$ is the diffuse intensity, and $\mathbf{L}_s$ is the specular intensity, as shown in Lecture 6, Slide 25. For the specular shading term, note that increasing k_s increases the intensity, and increasing p narrows the reflection lobe, i.e. yields a sharper lighting point, as shown in Lecture 6, Slide 27



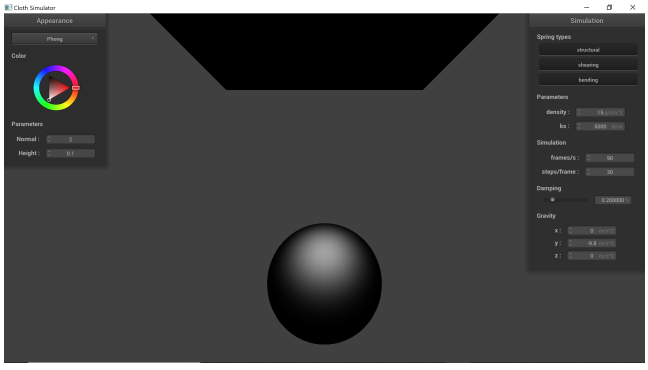
Ambient shading only

Ambient shading only



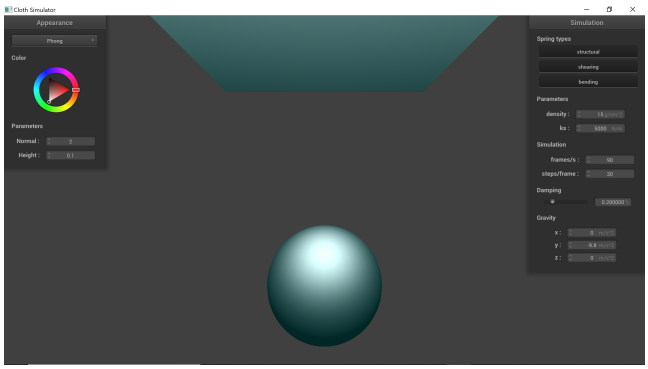
Diffuse shading only

Diffuse shading only



Specular shading only

Specular shading only

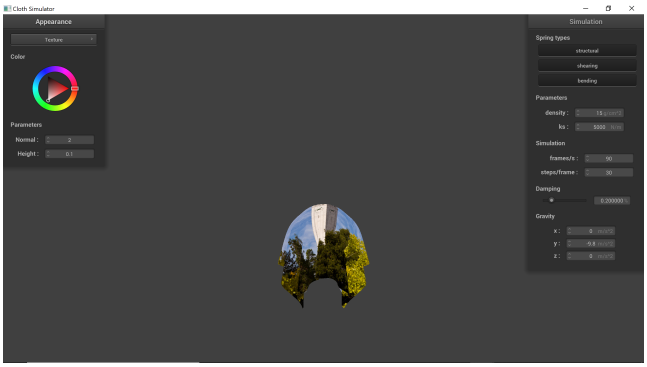


Blinn-Phong shading (all)

Blinn-Phong shading (all)

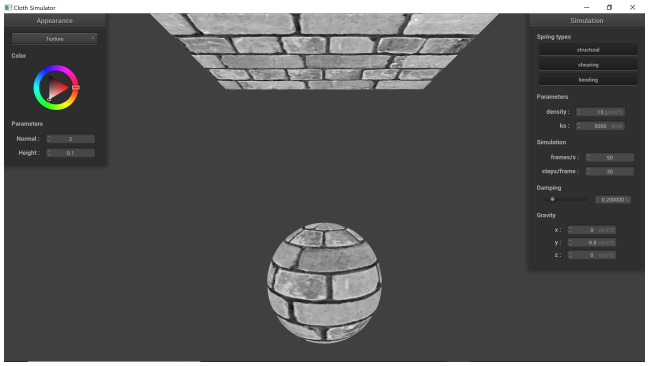
Task 3: Texture Mapping

out_color = texture(u_texture_1, v_uv)

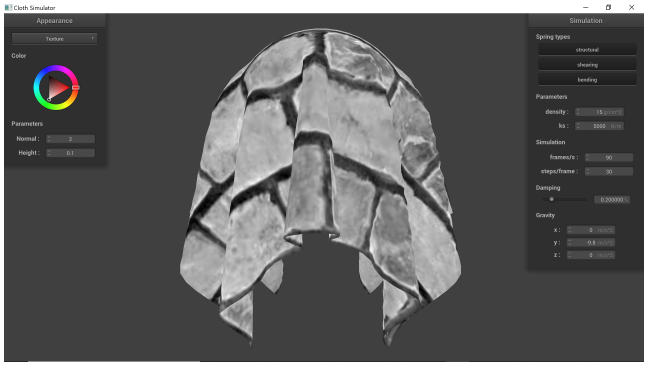


Texture mapping

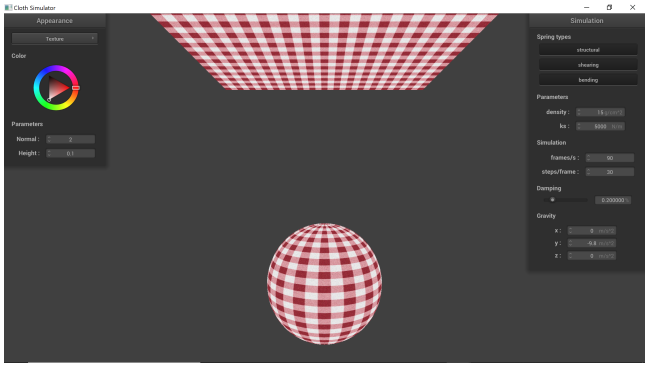
Texture mapping



Texture mapping



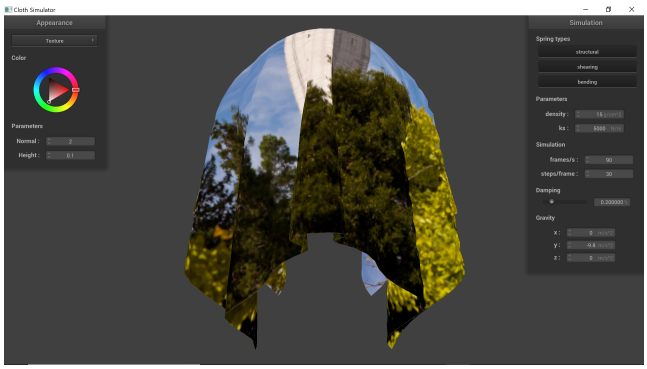
Texture mapping



Texture mapping

Texture mapping

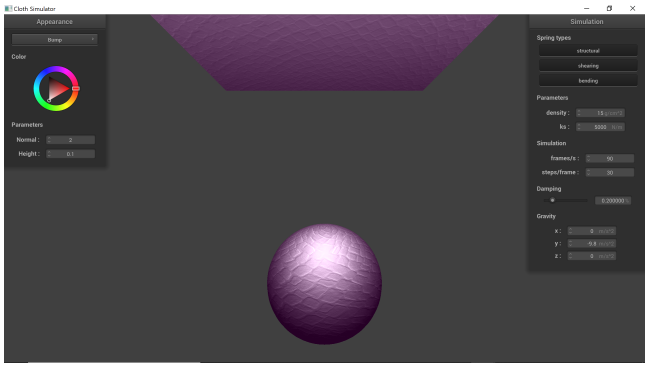
Texture mapping



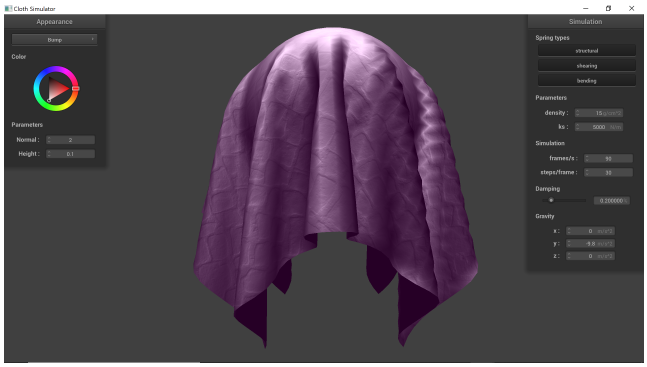
Texture mapping

Task 4.1: Bump Mapping

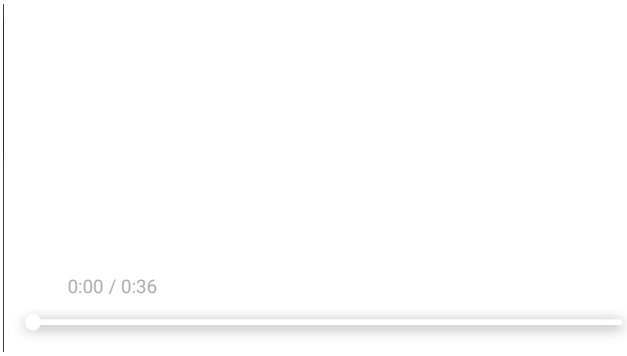
Bump mapping: modifying normal vectors of a mesh to give the illusion of detail.



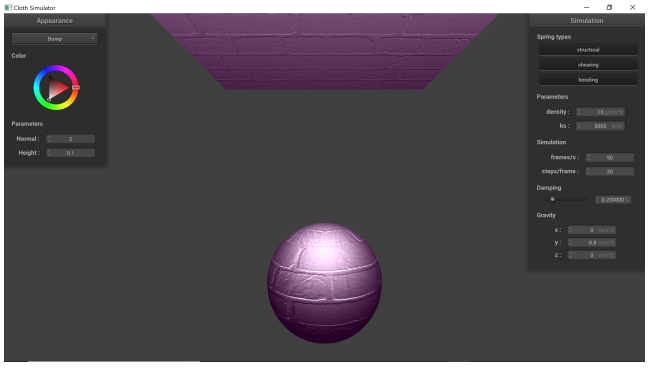
Bump mapping



Bump mapping



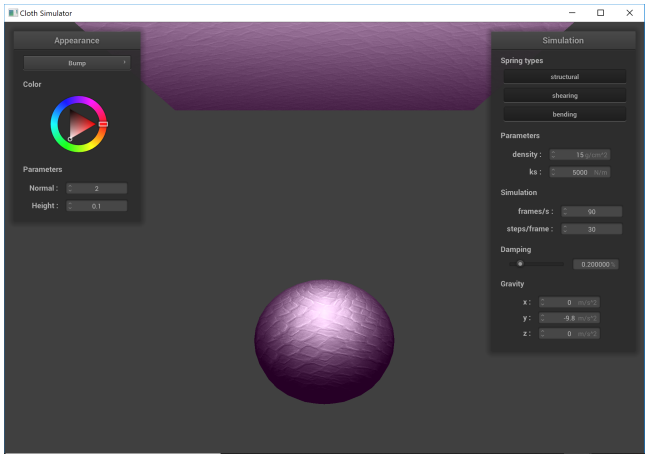
Bump mapping



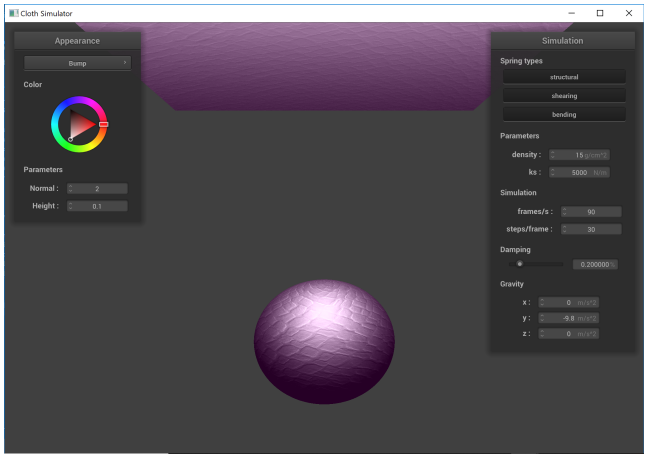
Bump mapping

Bump mapping

Above are images and videos of bump mapping using two different textures.



Bump mapping: -o 16 -a 16

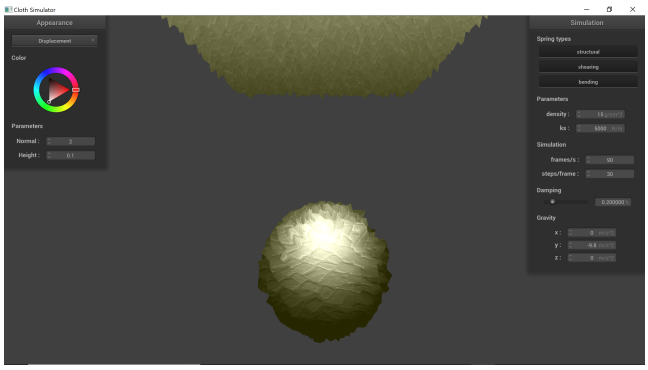


Bump mapping: -o 128 -a 128

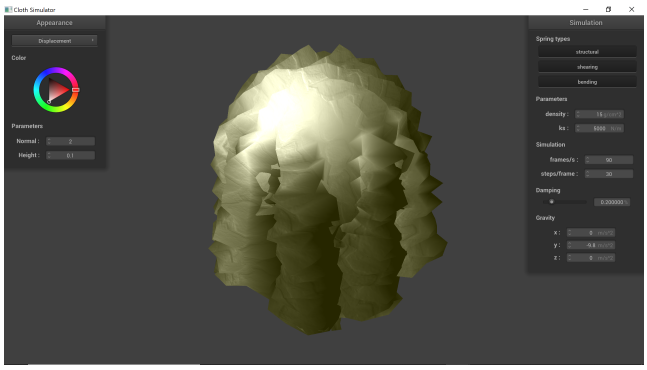
We can adjust the sphere's coarseness by setting the flags -o and -a. Using -o 16 -a 16, I can just barely see that the sphere's edges are slightly more pronounced than when using -o 128 -a 128. In the case of bump mapping, changing the coarseness of the sphere won't have drastic effects. However, changing the coarseness will have drastic effects for displacement mapping, to be discussed later.

Task 4.2: Displacement Mapping

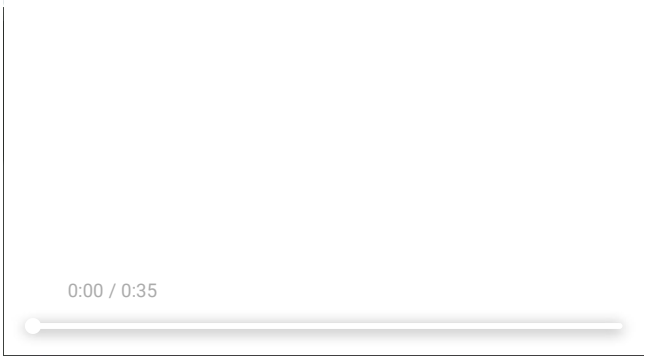
Displacement mapping: Bump mapping in addition to modifying vertex positions of a mesh to reflect the height map.



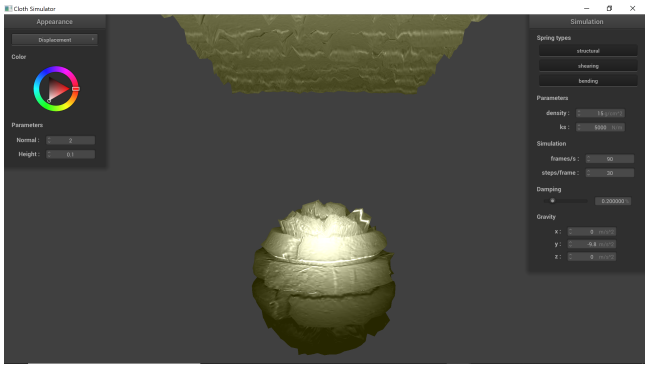
Displacement mapping



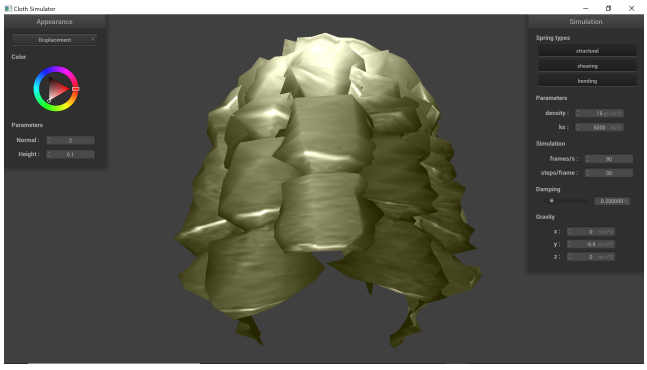
Displacement mapping



Displacement mapping

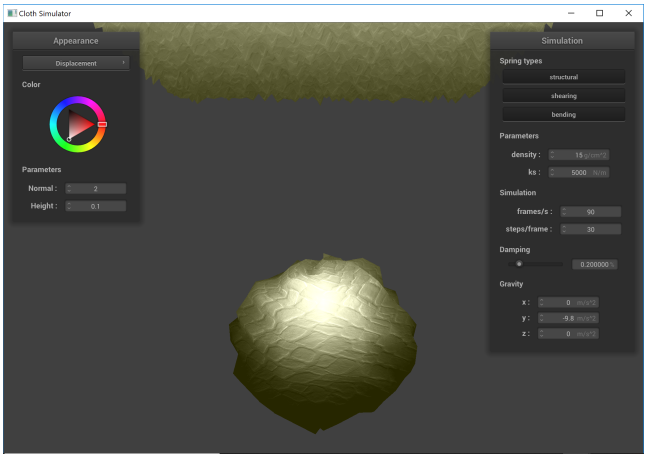


Displacement mapping

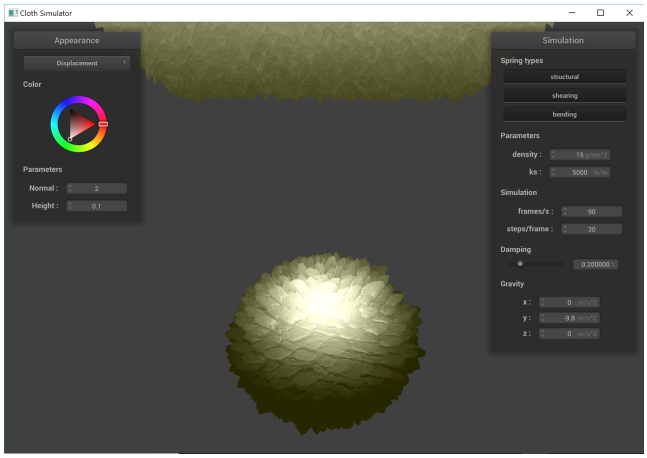


Displacement mapping

Above are images and videos of displacement mapping using two different textures.



Displacement mapping: -o 16 -a 16



Displacement mapping: -o 128 -a 128

We can adjust the sphere's coarseness by setting the flags -o and -a. Using -o 16 -a 16, I can see that there are much less protrusions than when using -o 128 -a 128. This makes sense, as -o 16 -a 16 results in a much coarser sphere compared to -o 128 -a 128.

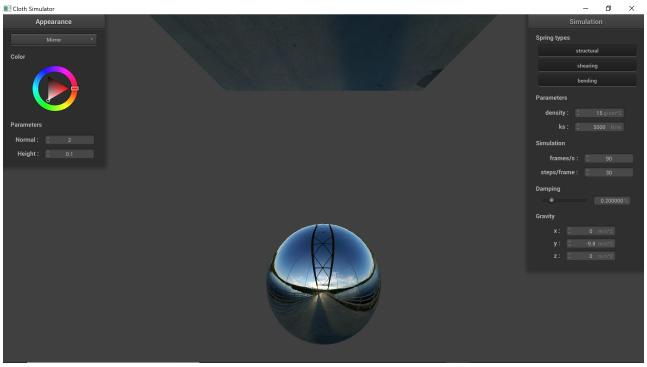
Task 5: Environment-Mapped Reflections

Process:

1. Calculate vec3 wo = u_cam_pos - v_position.xyz

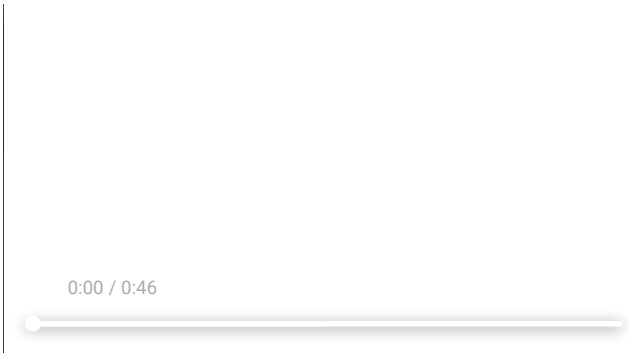
2. Calculate vec3 wi as shown in Lecture 14, Slide 16.

3. out_color = texture(u_texture_cubemap, wi)



Environment-mapped reflections

Environment-mapped reflections



Environment-mapped reflections