CS184: Computer Graphics and Imaging, SP21 Roman Taylor
Project 4: Cloth Simulation Zachary Young

Write-up

Overview:

In this project, we explore simulating cloth and lighting effects in shader programs. The cloth interacts with other objects, obeys Hooke's law, and falls with gravity. We will explore how changing various properties of our simulation like density and the spring constant affect the realism of our scene.

In the shader section we will implement diffuse and Blinn-Phong shading as well as texture mapping. As a bonus we created the Ukrainian flag in a shader program.

Part I: Masses and Springs

Implementation:

Building the cloth material required a small amount of organization and planning before implementing in order to accurately place the springs and point masses together. For the dimension of the cloth in `x` and `y`, we calculated the distance between each point mass as `width / (num_width_points - 1)` and `height / (num_height_points - 1)`. This places the first and last point masses correctly on the edge of the cloth in each dimension. We then placed the point masses depending on the cloth's orientation to produce realistic falling effects when it's standing on edge (e.g. adding random locations around the axis to produce more waves when falling). Finally, we placed the springs between the point masses by iterating through the point masses we just created then linking them depending on the spring's purpose.

Problems/Solutions:

The most challenging task was in adding the springs to the cloth. It was clear there were a number of edge cases to consider when iterating through the point masses. To alleviate this in a generalistic way, we decided to generate the indices for the sibling point mass without regard for the index's validity. This allowed us to then only place springs where valid indices exist by easily checking its validity.

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:

Roman Taylor
Zachary Young

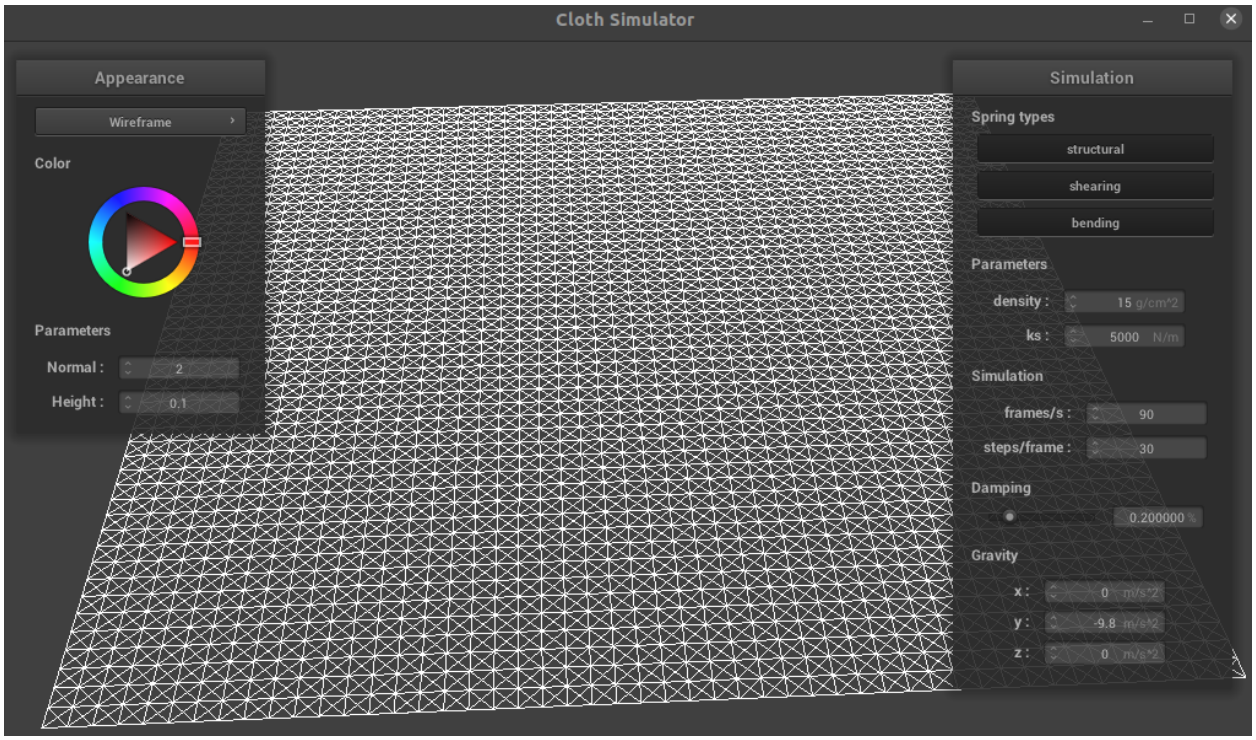


Fig 1: scene/pinned2.json
Cloth Wireframe

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

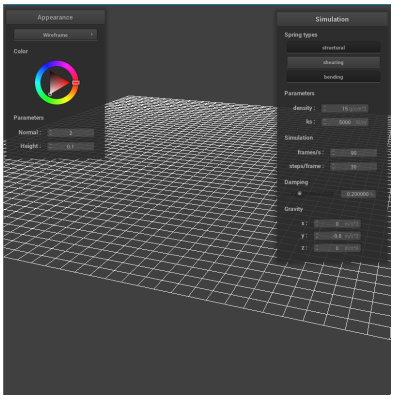


Fig 2: scene/pinned2.json
Cloth Wireframe (no shearing)

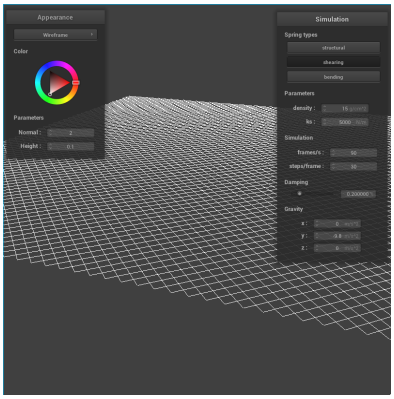


Fig 3: scene/pinned2.json
Cloth Wireframe (only shearing)

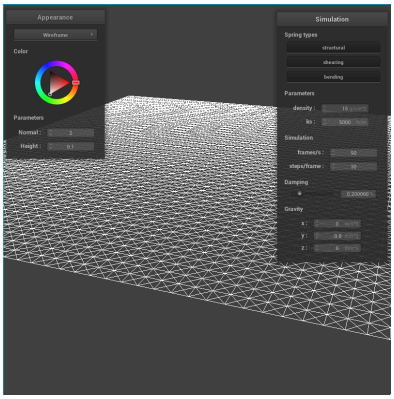


Fig 4: scene/pinned2.json
Cloth Wireframe (all constraints)

CS184: Computer Graphics and Imaging, SP21
Project 4: Cloth Simulation

Roman Taylor
Zachary Young

Part II: Simulation via Numerical Integration

Implementation:

In this part, we needed to add motion to our system of springs & masses. In order to do that, we had to calculate physical equations of motion and apply the correct amount of force to our system for each time step.

First, we calculated total force action on each point mass. This was done by calculating Netwton's 2nd law (external force) and applying it to every point mass. Then we calculated the spring correction force using Hooke's law (internal force) and applied it to each spring.

$$F = ma$$

$$F_s = k_s * (||p_a - p_b|| - l)$$

After calculating total force on each point mass, we computed new point mass positions using Verlet Integration and applied it to each point mass (unless the point mass was pinned).

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

Lastly, we referenced SIGGRAPH 1995 Provot paper to help us combat spring deformation. In short, we limited spring length to 10% above its resting state length at each time step.

Problems/Solutions:

The complexity of this question comes from applying math equations to our codebase. Because we are dealing with three dimensional vectors, it was challenging to visualize mathematics in this space which made debugging and sanity checks extra tricky. Fortunately, we did not run into any particular problem other than a variety of calculation mistakes that we solved by going over the math line-by-line and outputting the results of calculations for sanity checks.

Experiment with some of 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:

Roman Taylor
Zachary Young

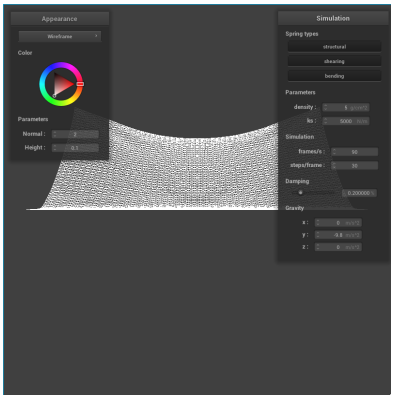


Fig 5: scene/pinned2.json
Ks = 5000
Density = 5

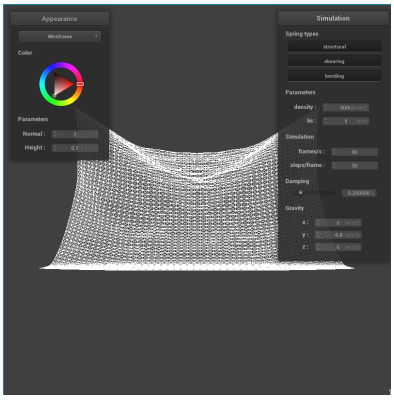


Fig 6: scene/pinned2.json
Ks = 5
Density = 5000

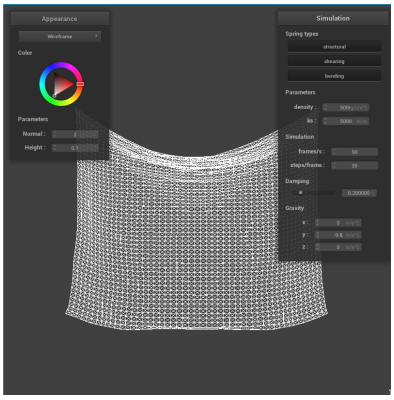


Fig 7: scene/pinned2.json
Ks = 5000
Density = 5000

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?

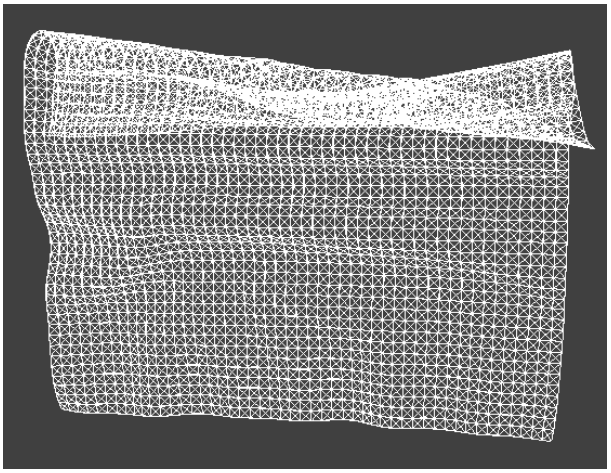


Fig 8: scene/pinned2.json
Density set to 0g/cm^2

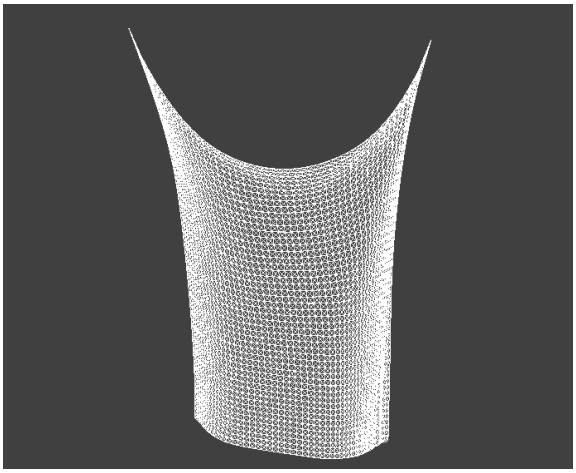


Fig 9: scene/pinned2.json
Density set to 50000 g/cm^2

Roman Taylor
Zachary Young

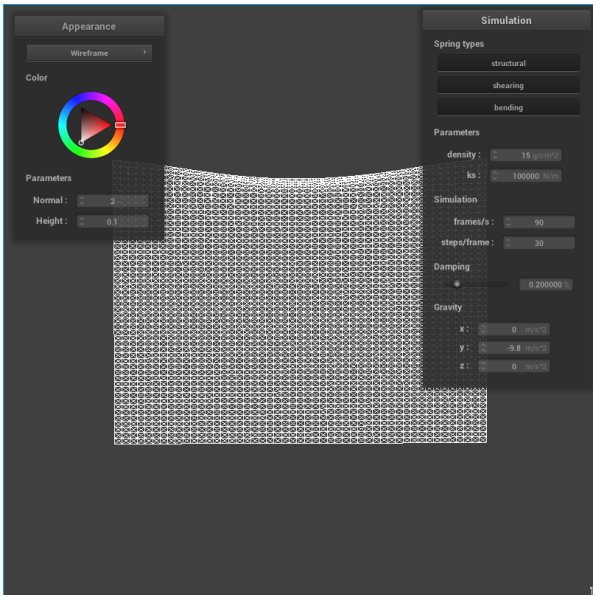


Fig 10: scene/pinned2.json
Spring constant set to 100000 N/m

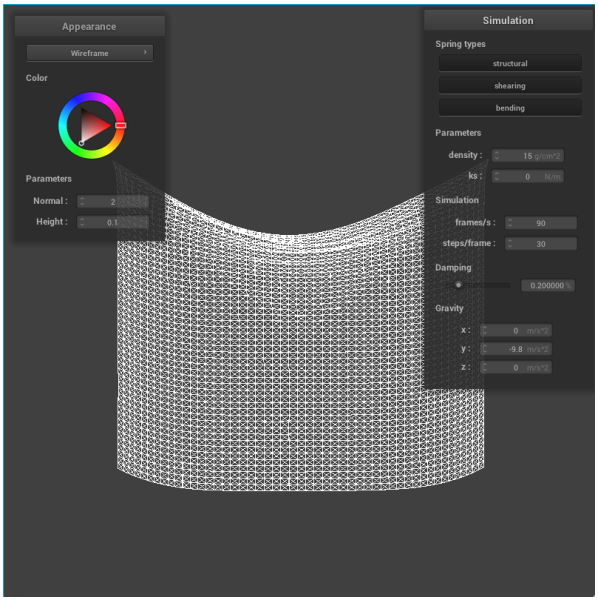


Fig 11: scene/pinned2.json
Spring constant set to 0 N/m

Spring constant (ks) changes the "crease" amount and intensity. A very high ks results in a relatively straight cloth with not many creases (see Figure 11). Low ks results in a significant droop between pinned corners due to low simulated spring tension (i.e. simulated forces between point masses). Increasing the spring constant results in less creases appearing on the cloth. Creases will increase in amount and intensity as we decrease the ks. When spring tension reaches really high levels (see Figure 10), spring tension overcomes the droop between pinned points resulting in a very tense (almost straight line) connection between upper corners and lack of creasing all over the fabric.

The fabric drops at almost the same rate despite different ks values. However, at low ks - cloth appears to have a "stretchy feel" on the way down and it takes some time for it to stop oscillating. Meanwhile at high ks - the cloth appears much more rigid and it does not "stretch/oscillate" nearly as much.

What about for density?

Density affects the mass of the cloth but this does not affect how fast the cloth falls. The effect of increasing the density, and thus its weight, of the cloth can be seen when comparing figures 13 and 16. In figure 16 the cloth has a density of 100 g/cm^2 and has more folds due to its weight than figure 13 which has a density of 1 g/cm^2.

What about for damping?

Roman Taylor
Zachary Young

High dampening makes the cloth fall significantly slower and decreases the amount of creasing. Little “waves” appear to consistently propagate through the cloth, at high damping rates, without ever stopping. Meanwhile, setting dampening to 0 results in a fast drop where the cloth never stops moving as if it is hanging in a high wind area.

For each of the above, observe any noticeable differences in the cloth compared to the default parameters and show us some screenshots of those interesting differences and describe when they occur:

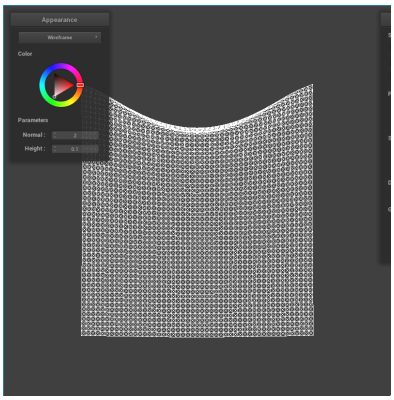


Fig 12: scene/pinned2.json
Default wireframe

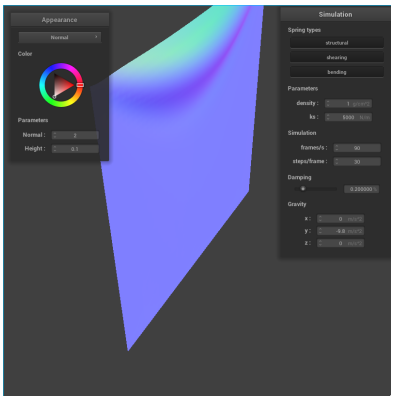


Fig 13: scene/pinned2.json
Density set to 1 g/cm^2

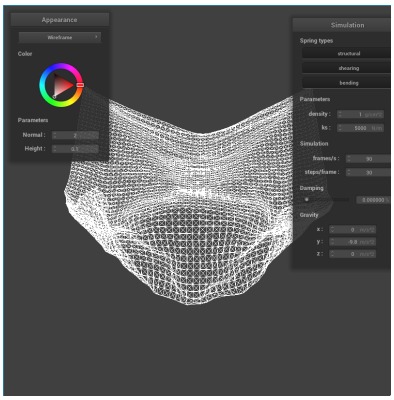


Fig 14: scene/pinned2.json
Damping set to 0.0

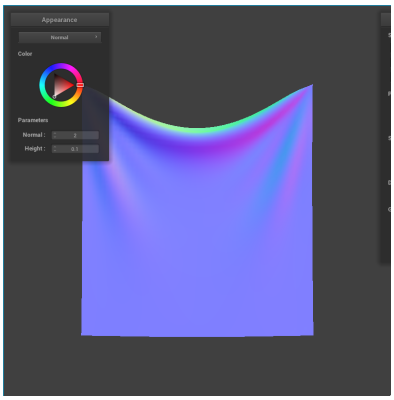


Fig 15: scene/pinned2.json
Default normal shading

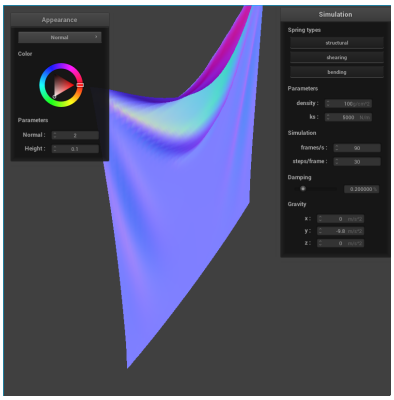


Fig 16: scene/pinned2.json
Density set to 100 g/cm^2

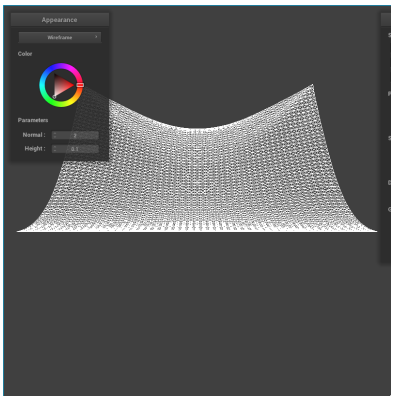


Fig 17: scene/pinned2.json
Damping set to 1.0

Additionally, Changing the damping value affects the amount of velocity to keep in the system on each time step. With low damping there is a lot of remaining velocity from the previous time step causing the material to deform in more complex ways. Increasing the damping smooths out these deformations. This can be seen when comparing figures 14 and 17.

Additionally:

- For Density: Please refer to figure 8-9 above
- For Spring Constant: Please refer to figure 10-11 above

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:

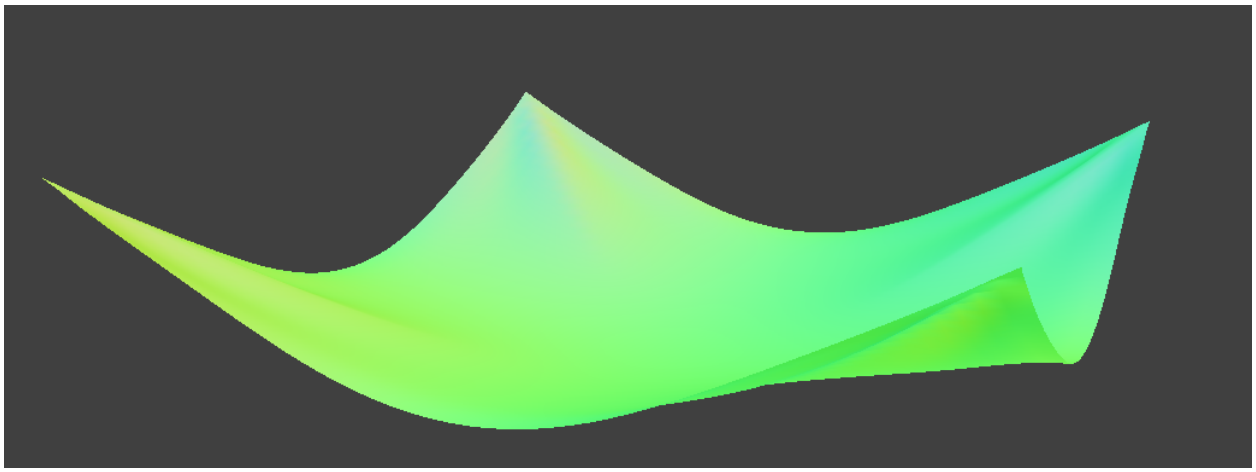


Fig 18: scene/pinned4.json
Resting state & default parameters

Part III: Handling collisions with other objects

Implementation:

Implementing collision between our cloth, other objects, and itself is straightforward with a lot of compromises. Since we can reason about collisions on our cloth on a point mass by point mass basis we can utilize the same tools we've learned from previous projects to implement collisions for our cloth.

For collision with a sphere, it was simply a matter of determining if a point mass is within the sphere using the distance from the sphere's origin to the point mass's position. If the distance between those is less than the radius of the sphere, then the point mass is within the sphere. We simply move the point above the sphere by interpolating to a position above the sphere's surface.

Now we had to account for the collisions with planes. This was done by detecting a point mass crossing the plane -- in the previous time step -- and moving it's current position to a new position to the other (original) side of the plane. We calculate the new position by taking the previous time step position and finding its intersection point with the plane. Then we calculate a correction vector which can move the last position to the intersection point (small correction is applied such that we do not actually touch the surface). Lastly, we set our position to be the

corrected last position scaled down by friction.

Problems/Solutions:

The most frustrating part of this section is in the inaccuracies of the simulation, particularly when the cloth is at rest on the plane. In order to keep the cloth above the plane you must first let the cloth move through the plane then correct the error. This requires you to make a choice about where to move each point mass when it has passed through the plane. The first solution is to linearly interpolate on the point mass's current trajectory to place it above the plane. The second solution is to place the point a specified distance away from the plane directly above the point of intersection. Both of these result in the point mass moving on every iteration upon reaching its resting state even though it can no longer move through the plane. The third solution is to simply stop correcting the point mass's position once it has moved through the plane. Simply assign the new position to the old position. The point mass will continue on its trajectory in the next time step resulting in the same outcome. This is our chosen solution.

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:

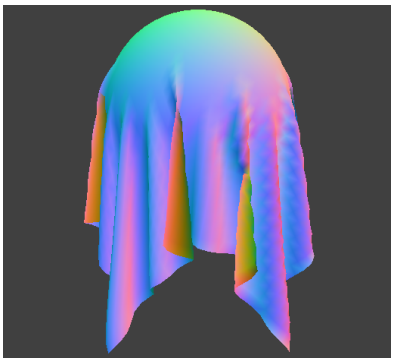


Fig 19: scene/sphere.json
Ks = 500

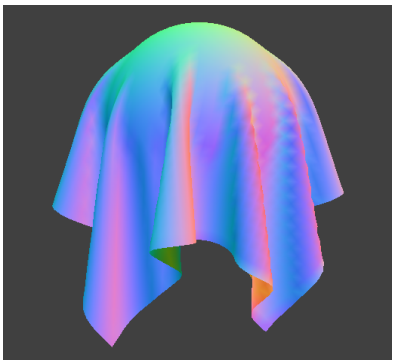


Fig 20: scene/sphere.json
Ks = 5000

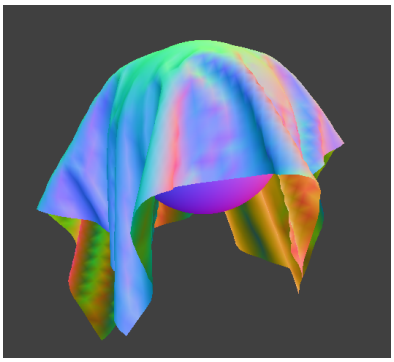


Fig 21: scene/sphere.json
Ks = 50000

Describe the differences in the results:

The spring constant Ks determines how resistant our spring is to change is in the direction of the axis of its helix. Lower values of Ks make our springs less resistant to change while higher values of Ks make our springs more resistant to change. This can be seen in figure 19 through 21 where figure 19 has a more relaxed appearance and figure 21 has a more tight or scrunched appearance.

Since the force between the point masses at the ends of each spring is calculated using Hooke's law, we can also interpret the Ks value as how much the

spring wants to return to a resting state.

Show us a screenshot of your shaded cloth lying peacefully at rest on the plane:

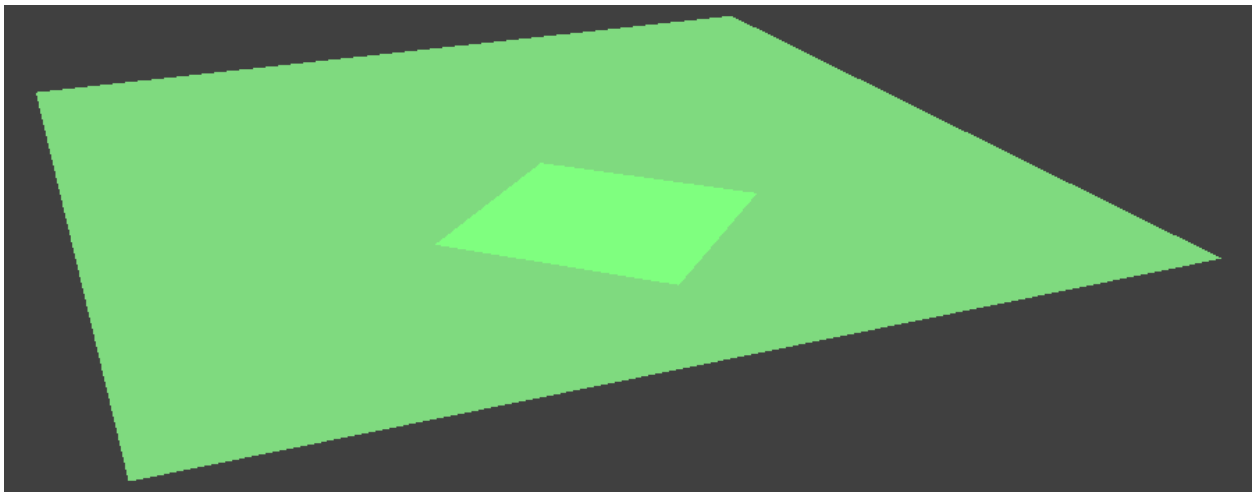


Fig 22: scene/plane.json
"Normal" Shaded Cloth lying at rest

Part IV: Handling self-collisions

Implementation:

Self-collision problem boils down to calculating (and adjusting if needed) distances between all pairs of points which is relatively easy. Unfortunately this is too computationally heavy for a typical computer. The brute force algorithm will run painfully slow.

To speed up self-collision detection, we implemented a spatial hash map by placing every point mass into a 3D rectangle represented by a simple numerical value. After subdividing the area, for each pair within the same 3D rectangle, we found the distance of a given pair. If the distance was below our threshold, we calculated & applied a correction vector to our point mass such that it moved away from it's companion. This method enabled us to significantly reduce calculation overhead.

Problems / Solutions:

The most difficult task in implementing collisions on our cloth was in correctly calculating internal collisions while considering the computational cost of using a large amount of point masses. This requires an efficient hashing function to

Roman Taylor
Zachary Young

divide the scene into a system of globally positioned cubes. This means we only need to test for collisions within the confines of each cube. Two hash functions comparable in realism we found were $7x + 5y + 3z$ and $x^6 + y^4 + z^2$ where x, y, and z are the cube index in each dimension. The key to self collisions is in using the absolute value of the position of each point mass. The second equation accounts for this by using even numbered exponents to create a unique distribution.

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):

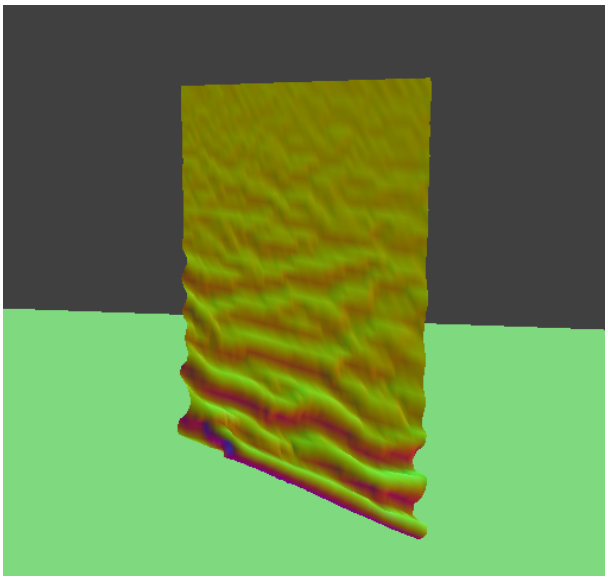


Fig 23: scene/selfCollision.json
Early state

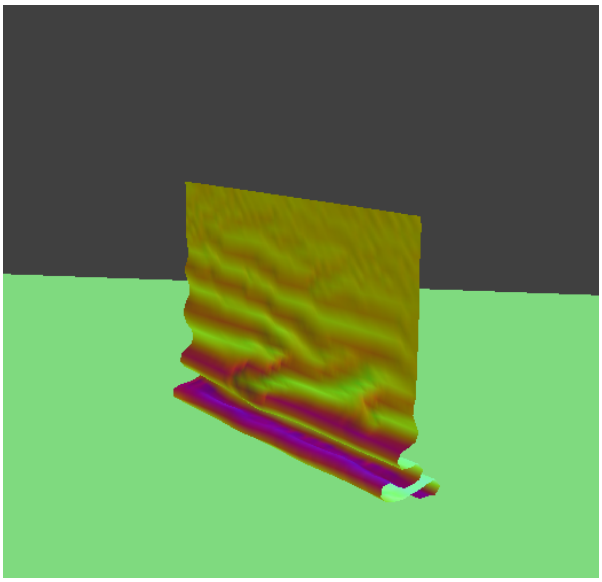


Fig 24: scene/selfCollision.json
Intermediate state

Roman Taylor
Zachary Young

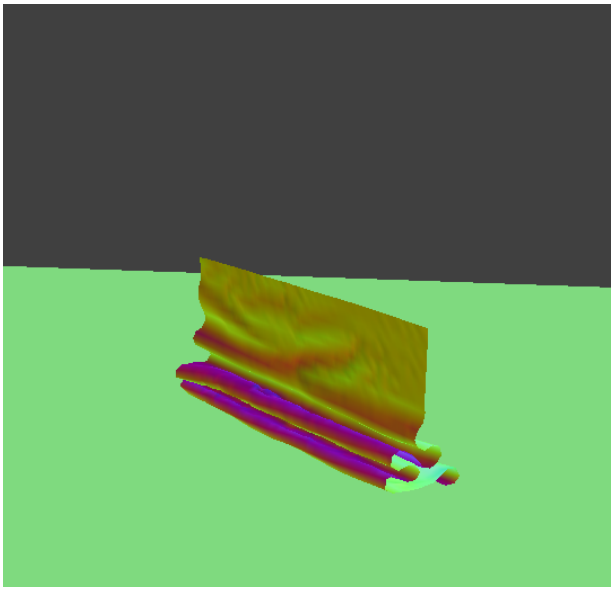


Fig 25: scene/selfCollision.json
Late state

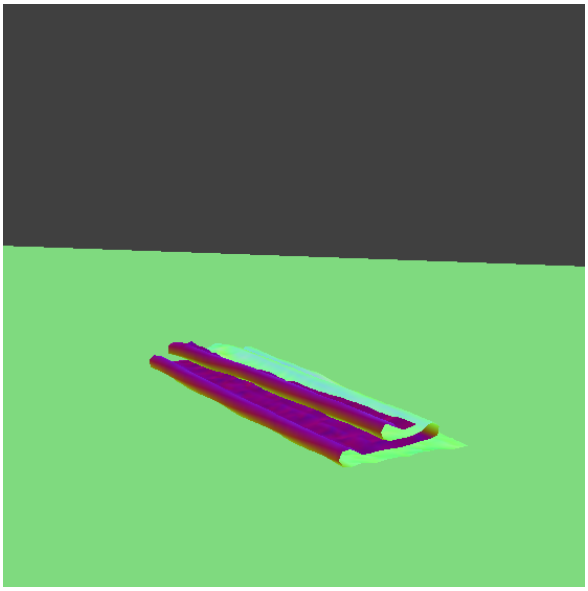


Fig 26: scene/selfCollision.json
Nearly restful state

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:

Density:

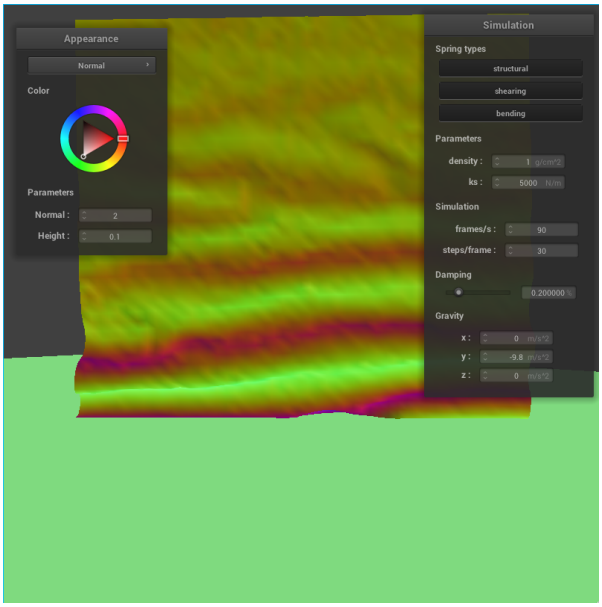


Fig 27: Density 1 g/cm^2
20th frame

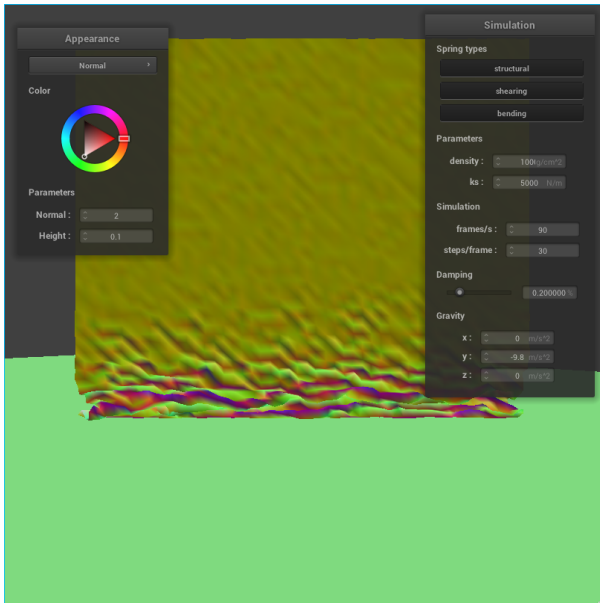


Fig 28: Density 1000 g/cm^2
20th frame

Since density affects the weight of the cloth as well as the elasticity, it can

be seen by comparing figures 27 and 28 that higher densities produce more complex wave patterns. Why would density affect the springs? Denser materials tend to be unable to flex as much.

Ks:

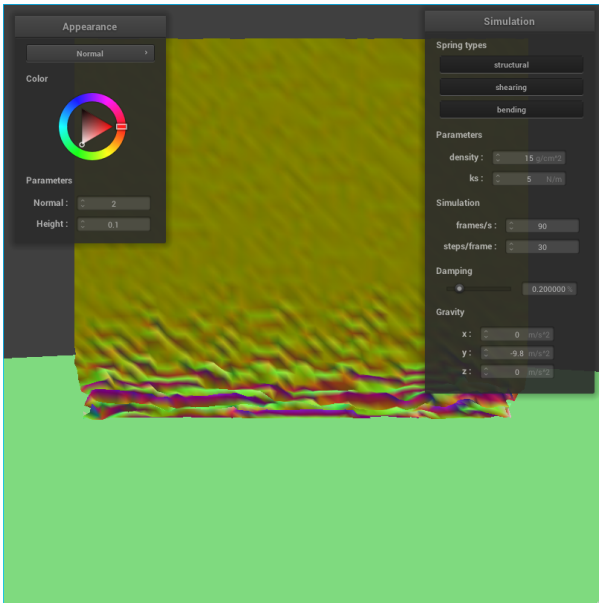


Fig 29: Ks: 5 N/m
20th frame

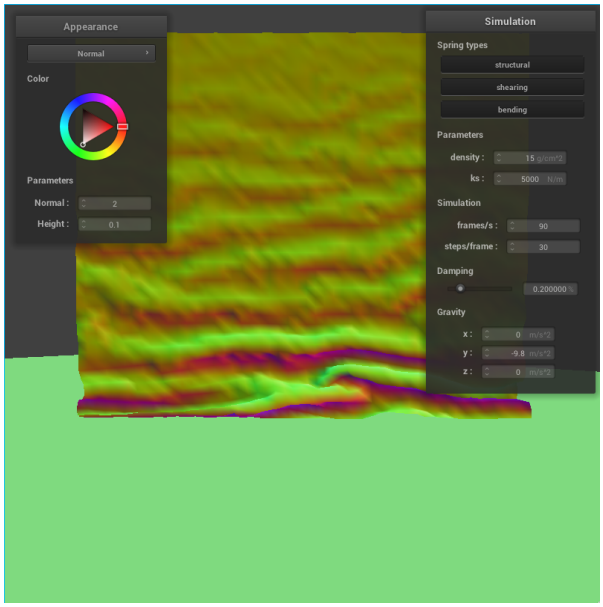


Fig 30: Ks: 5000 N/m
20th frame

Since Ks is the spring constant, changing this will determine how relaxed the cloth appears. This can be seen in figures 29 and 30 where the higher Ks value produces less complex waves in the cloth due to the spring forces being higher.

Though the spring constant can be decoupled from the density to produce a wider range of simulated materials, this project focuses on cloth-like materials so it makes sense to decrease the spring constant when the mass increases simulating a situation where the material gains mass and loses flexibility.

Part V: Shaders

Implementation:

Shader programs are compiled parallel Graphics Processing Unit (GPU) programs that allow you manipulate data such as vectors and matrices on the GPU. The implementation of shader programs can be as simple or complex as you can imagine but simulating physics of the real world is implemented in the same manner as we have done in the past on the CPU. The diffuse shader was simply to color a vertex based on the

direction of its normal and the direction of the light. Blinn-Phong shading combined diffuse shading with ambient shading and specular highlight utilizing the same techniques. Texture mapping and environment-mapped reflections were as simple as calling a single function to grab a pixel from a texture stored in the GPU. Bump mapping required changing the normal of each vertex based on the value stored in a texture. This imprinted the texture onto the surface while still applying the correct lighting details. Displacement mapping was simply applying the same type of normal manipulation as bump mapping but in the vertex shader to affect the vertex position based on the texture. And finally, we also created a custom shader that can be used to render simple multi-color flags such as Russian, German, and Italian.

Problems / Solutions:

We faced a few problems while implementing shader programs for our cloth simulation. The first was in converting between data types. The GLSL language is very flexible in how you can extract data but unforgiving in how you present that data to the program. To help alleviate our casting woes, we consulted various OpenGL GLSL documentation websites. The next problem we faced was in implementing bump mapping in combination with a lighting model. The same vector math problems popped up like unnormalized vectors and difficulty switching between coordinate spaces.

The overarching theme of most shader program problems we faced were in interpreting the problems in our math given an incorrect output. With some textures it was easy to spot an incorrect orientation but lighting effects are much more difficult to determine their accuracy with so many variables available.

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

Shader is a program that runs on your Graphics Card instead of your Central Processing Unit. It takes input from your program, performs the calculations, and returns the result. In short, shaders enable you to speed up the calculational overhead within your software by allocating the work to another piece of hardware.

There are two primary types of shaders: vertex and fragment. Vertex shaders read and write all per-vertex values. This enables vertex shaders to move / apply transformations to vertices. Fragment shaders write per-pixel values and are typically used to calculate & fill-in the given fragment's color. Fragment shaders return a single four dimensional vector.

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.

The Blinn-Phong shading model combines three lighting techniques to create realistically lit objects. The first component is the ambient lighting which gives every vertex the same color. This results in a 2-dimensional looking object which can be seen in figure 31. The second component is diffuse shading which colors the vertex based on which way its normal is facing. If the normal is facing the light source, it receives all the light. The amount of light decreases until the normal faces completely away from the light source (i.e. it's on the dark side of the object). An

Roman Taylor
Zachary Young

example can be seen in figure 31. The final component which increases the realism dramatically is the specular highlights. The highlights are calculated the same as diffuse shading but utilize the half angle between the light vector and camera vector. To give it a focused appearance the angle between the normal and half angle is exponentiated which reduces the angle at which vertices will be lit by the light source. Simply add these three components up to finish the model seen in figure 31.

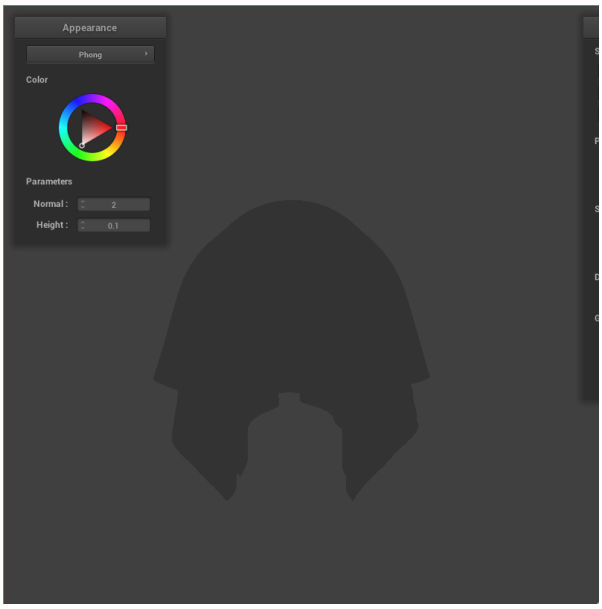


Fig 31: Blinn-Phong, only ambient

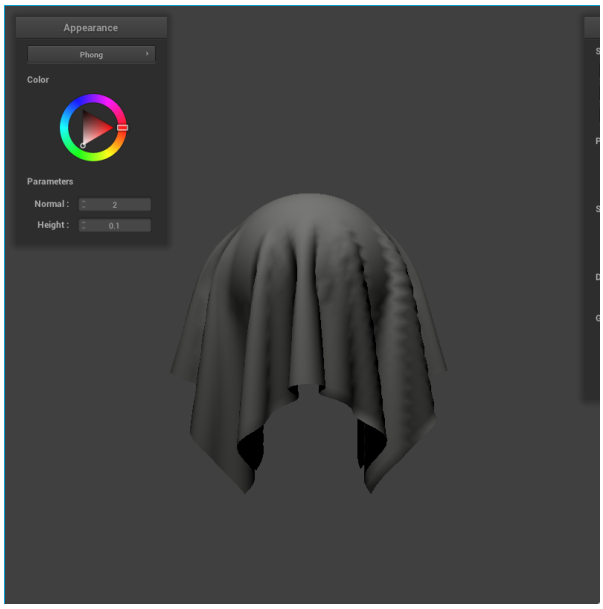
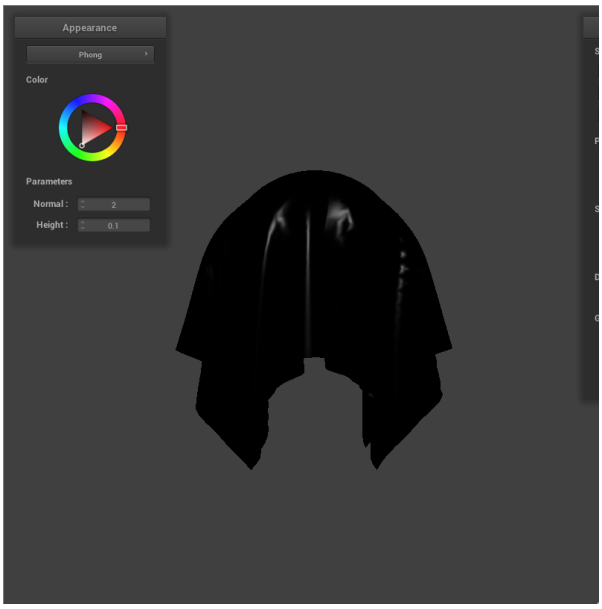
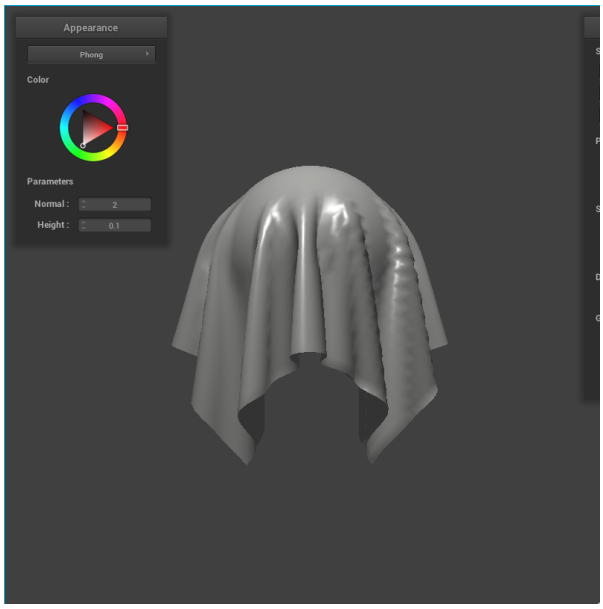


Fig 32: Blinn-Phong, only diffuse





Roman Taylor
Zachary Young

Fig 33: Blinn-Phong, only specular

Fig 34: Blinn-Phong, complete

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

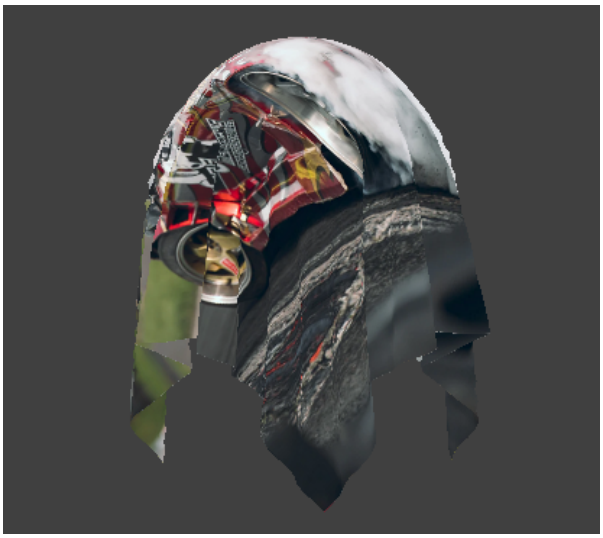


Fig 35: scene/sphere.json
Custom Texture Applied[1]

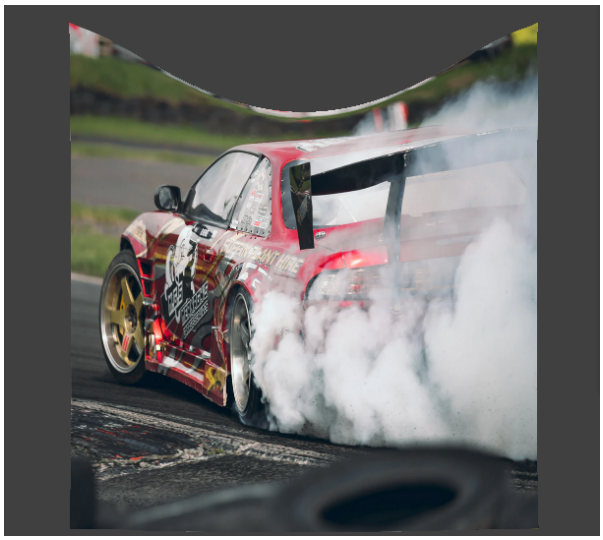


Fig 36: scene/pinned2.json
Custom Texture Applied

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.

Bump mapping and displacement mapping are two techniques that allow us to mimic complex surface textures to simulate various materials. Bump mapping is implemented in a fragment shader and calculates a new vertex normal based on the supplied texture. This results in a flat appearance at closer inspection which can be seen in figures 37, 38, and 39 (like it was printed on the surface). Displacement mapping adds a vertex shader to the bump mapping fragment shader that manipulates the vertex position based on the supplied texture. This results in a surface deformation that takes the shape of the supplied texture which can be seen in figure 40.

[1] Fig 32 Texture Photo Reference - https://unsplash.com/photos/H7ukvaVGPqg

Roman Taylor
Zachary Young

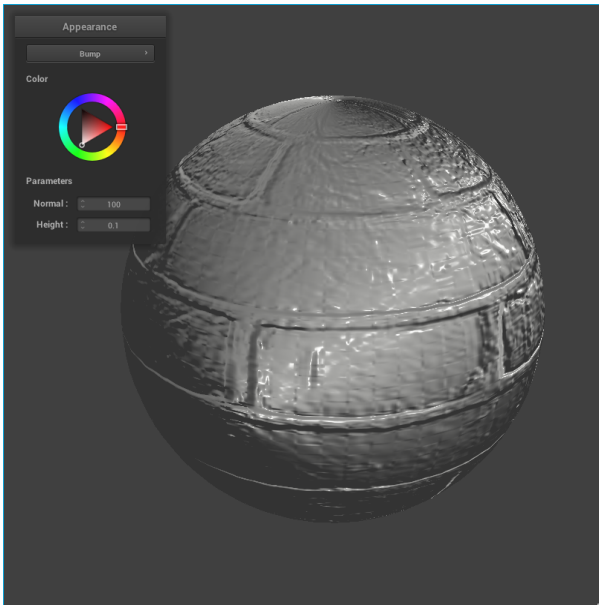


Fig 37: texture_3.png
Bump mapping sphere

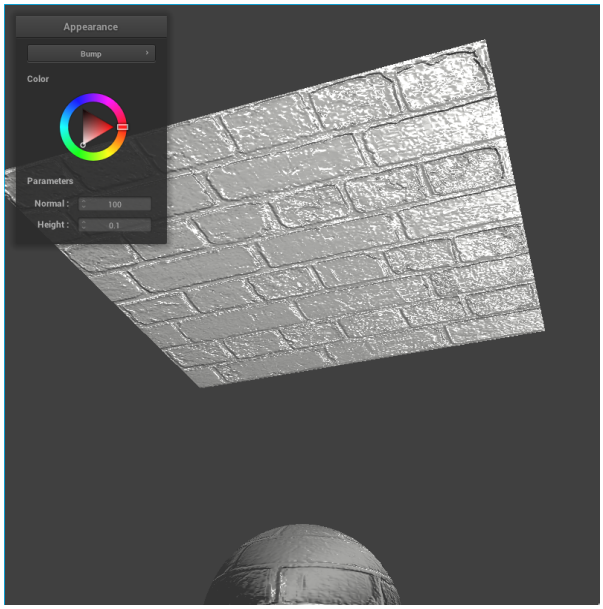


Fig 38: texture_3.png
Bump mapping cloth from below

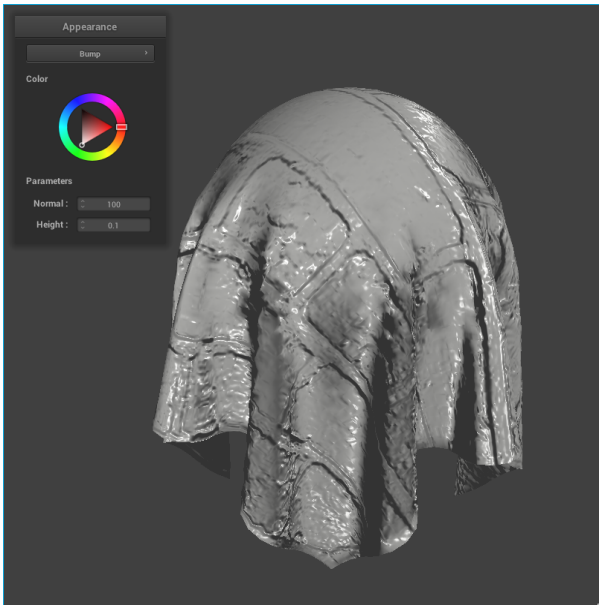


Fig 39: texture_3.png
Bump mapping cloth on sphere

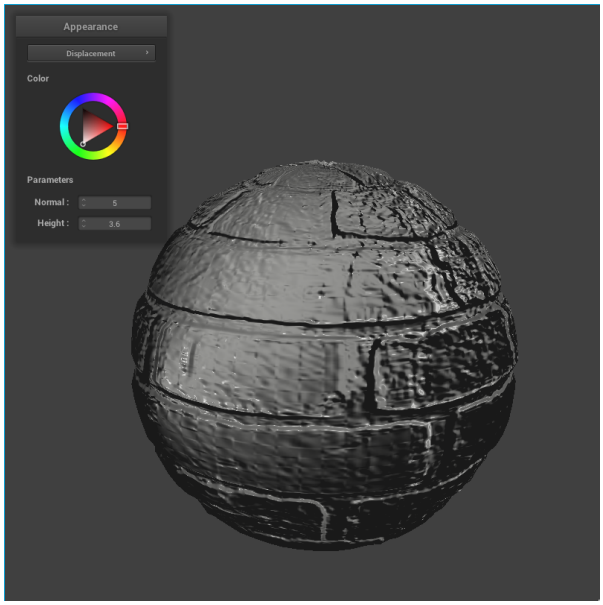


Fig 40: texture_3.png
Displacement mapping sphere
(note deformed surface)

Compare how 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.

Using a different mesh resolution can increase the accuracy of the displacement

mapped textures dramatically. From our tests it appears increasing the sphere resolution does not provide any benefit for a bump mapped texture. This is likely due to the texture correctly mapping to the surface. But for displacement mapping, the increase in sphere resolution has dramatically increased the realism of the texture. The deformations happen in the correct locations on the higher resolution sphere due to there being more vertices for the GPU to interpolate between. Figure 43 and 44 show how increasing the sphere resolution increases the accuracy of the displacement mapping. Note that the fragment shader works the same in both bump and displacement mapping at any sphere resolution.

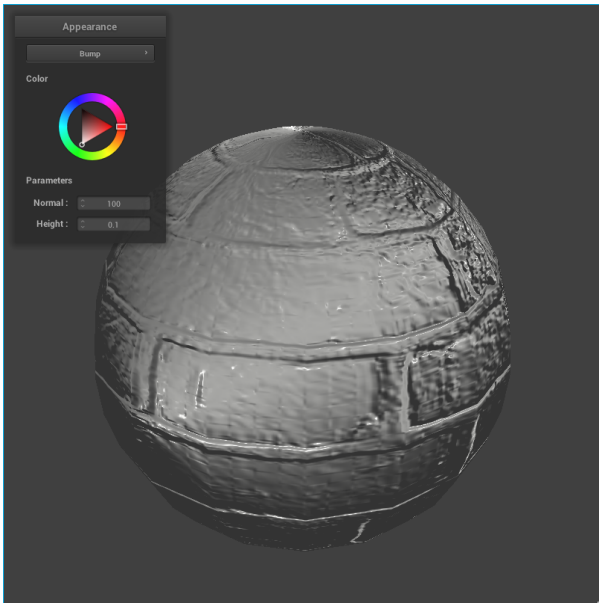


Fig 41: texture_3.png
Bump mapping sphere
Resolution 16

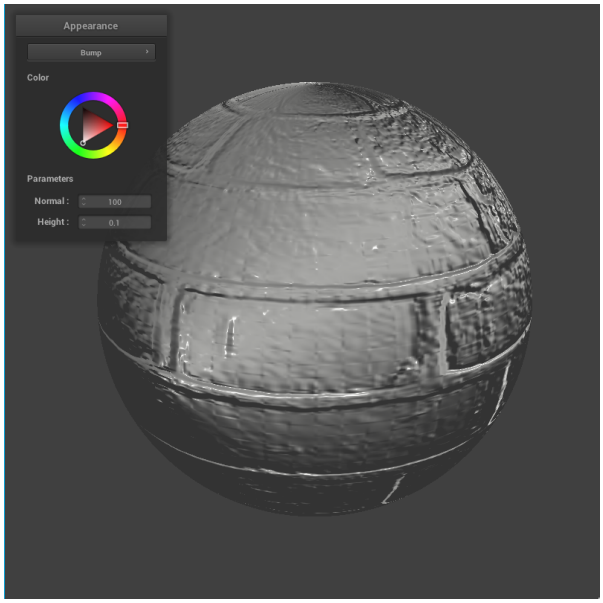


Fig 42: texture_3.png
Bump mapping sphere
Resolution 128

Roman Taylor
Zachary Young

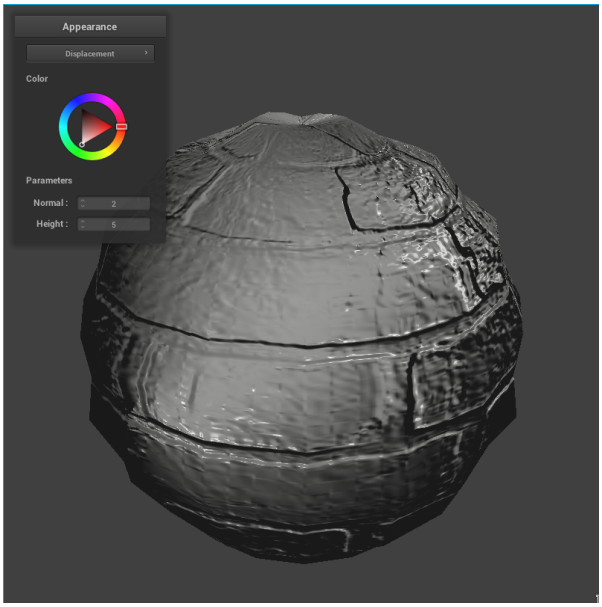


Fig 43: texture_3.png
Displacement mapping sphere
Resolution 16

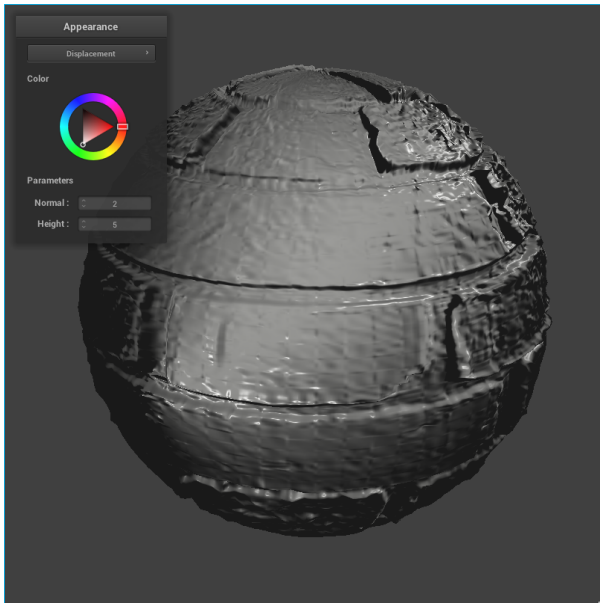


Fig 44: texture_3.png
Displacement mapping sphere
Resolution 128

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

Fig 45: scene/sphere.json
Mirror shader sphere

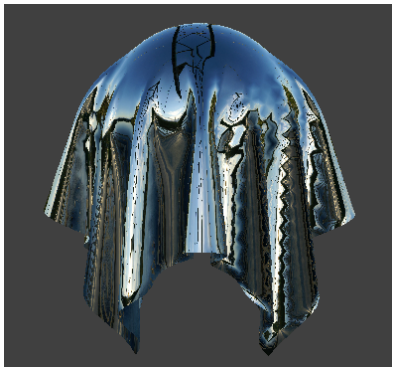


Fig 46: scene/sphere.json
Mirror shader cloth

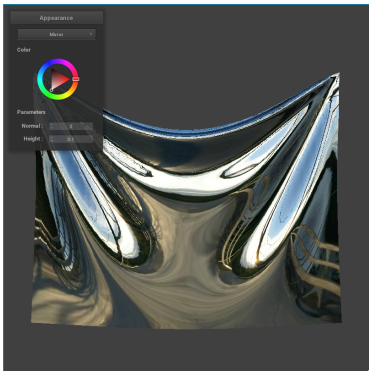


Fig 47: scene/pinned2.json
Mirror shader cloth

[EXTRA CREDIT: Simple Flag Shader] Explain what you did in your custom shader, if you made one.

We have created a custom shader that can be used to render simple horizontally shaded flags such as Ukranian, Russian, and German. With a simple axis modification, it could be adopted to render vertically shaded flags such as Italian and French.

Roman Taylor
Zachary Young

Since our vertex coordinates are normalized to the world coordinates we simply select the pixels based on their position and color them accordingly (i.e. `if vertex.y > 0.5`).

This enables us to independently color parts of our scene based on the `percent height` & `percent width` location on our screen. (see fig 48)

Additionally, we've combined our custom colors with diffuse shading for a more realistic look. (see fig 49)

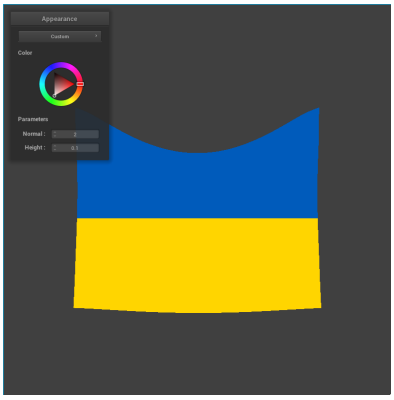


Fig 48: scene/pinned2.json
Custom shader - Ukranian Flag
No Diffuse Shading

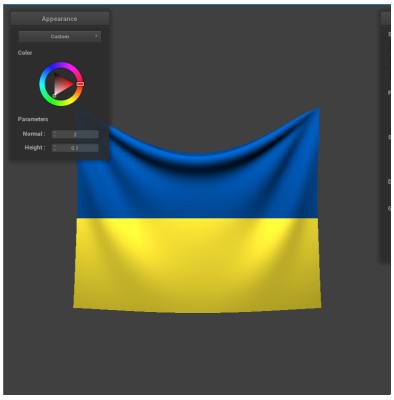


Fig 49: scene/pinned2.json
Custom shader - Ukranian Flag
With Diffuse Shading

Final Thoughts: Collaboration

How we collaborated, how it went, and what we learned:

As in previous projects, we have worked concurrently on different branches while sharing concepts, thoughts, and helping each other debug in real time. This enabled us to work through all of the concepts independently while enjoying the benefits of team collaboration at the same time.

So far we believe this to be the best approach for CS184 projects and we rarely deviate from our strategy.

A curious benefit of using our multi branch method is gaining an insight of how another person may turn an algorithm / instructions into code. Typically, we would end up with very similar code. But sometimes, our methodology would diverge and we would end up with interesting implementation differences that ultimately reach the same goal. Additionally, this is an excellent way to practice git branch management commands rather than only working with a master branch.