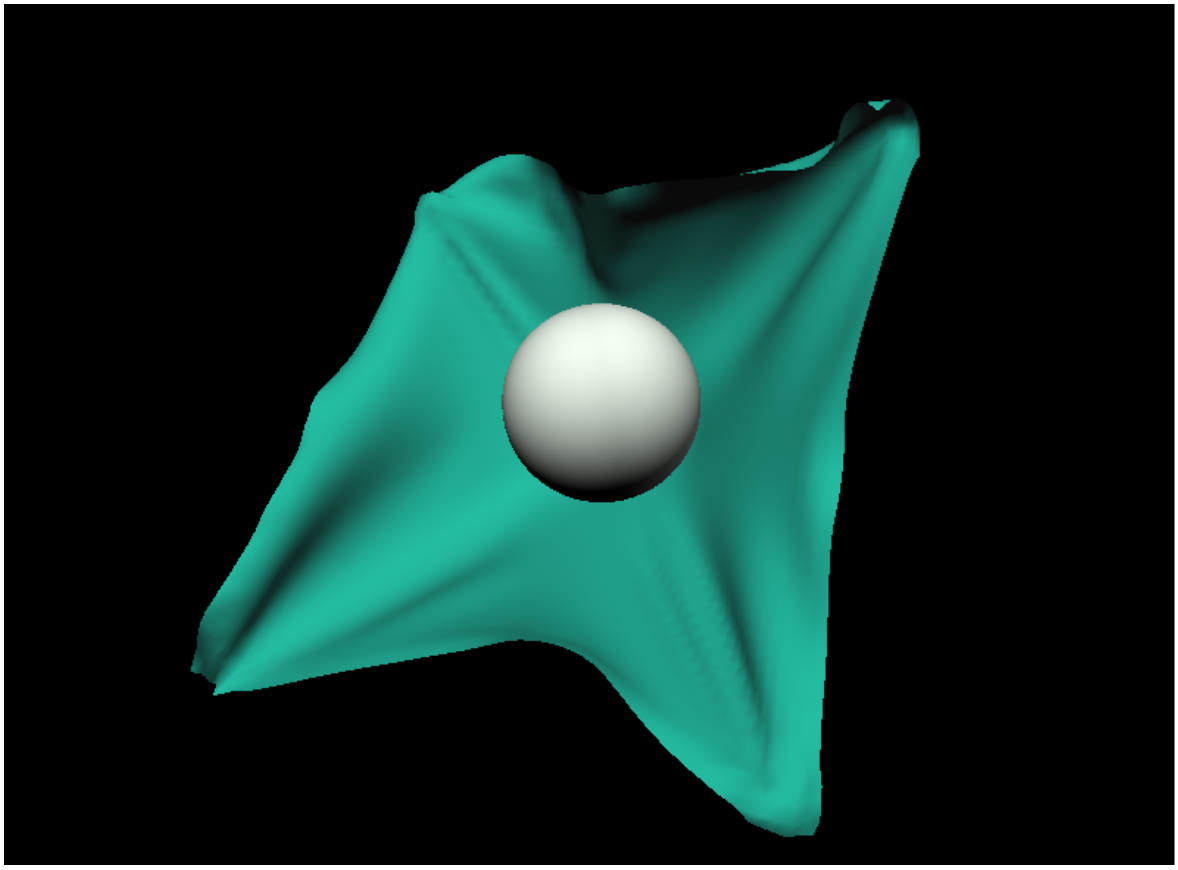
CS184 Final Project: Cloth Simulation

Peterson Cheng (cs184-bj) Parker Mossman (cs184-cz)
Eric O'Neill (cs184-eg) Rachel Wu (cs184-cg)

May 14th 2013

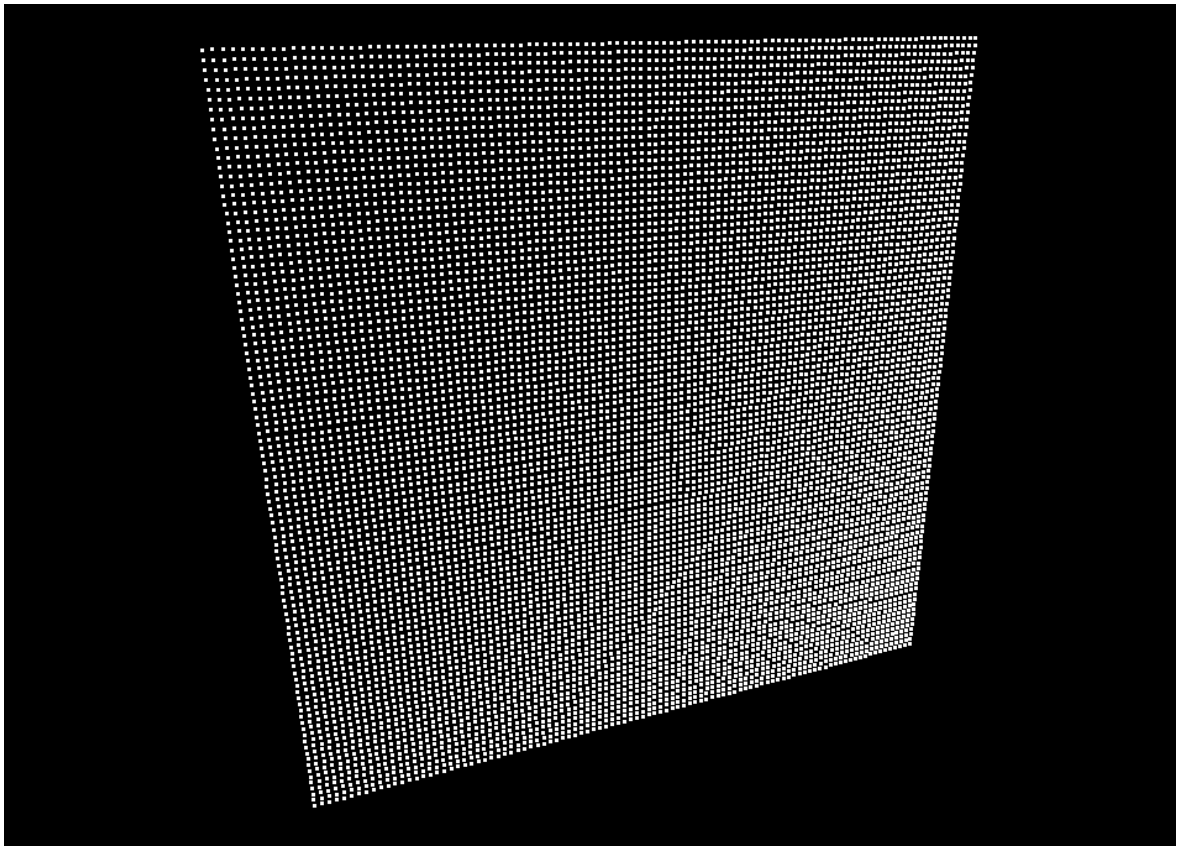


1 Introduction

We implemented a cloth simulation using a mass constraint system over a grid of particles. The mass constraint system involves determining a force vector for each particle in the grid, and calculating a resulting acceleration using Newton's 2nd Law. We then calculate the particle's new position through Verlet integration, and finally place a series of constraints on nearby particles to enforce a cloth-like rigidity on our grid of particles. External forces such as gravity and wind allow for a more realistic simulation of our cloth. This cloth also correctly interacts with spheres, and the user can manually move the sphere using the keyboard. Additionally, the cloth can be dragged, stretched, and pinned in real time with mouse click-and-drag interaction. We used OpenGL to facilitate all shading, drawing, transformations, and interactions with our cloth.

2 Representation

We begin by representing the cloth as a two-dimensional grid of particles – it's stored as a one-dimensional vector of particles in the cloth object. Each of these particles knows its instantaneous acceleration vector and position, and a base force acting on all particles (i.e. gravity). We have a boolean toggle for fixed/nonfixed particles, and an addForce method which updates the particle's acceleration given a force vector according to Newton's Second Law. At this stage, we render each position of the particle as a point, to get a simple grid representation of the cloth.



3 Basic Simulation

Now that we have the representation of our cloth, the next step is getting this cloth to move in a realistic way. To do so, we want to calculate a new position for each particle in our cloth. This is where we implement our particle class's *updatePos* method

```
updatePos(vec3 delta) {
    pos += delta;
}
```

We now want to calculate this delta, and since each particle knows its acceleration, we want to integrate to find a position. We do this using Verlet Integration[2], a scheme in which a new position x' is calculated based on a particle's current position x, acceleration a, and previous position x^*. The integration equation thus looks like

$$x' = 2x - x^* + a(\Delta t)^2$$

$$x^* = x$$

Now we just define Δt as a variable called *timestep*, and include a damping term to represent forces such as friction and air resistance. Now we can implement our full *particleTimeStep* method as follows

```
particleTimeStep() {
    vec3 temp = pos;
    updatePos( (pos-old_pos)*(1-damping)+(accel*timestep));
    old_pos=temp;
    resetAccel();
}
```

The call to resetAccel() returns a particle's acceleration vector to the base acceleration. This prevents forces and accelerations accumulating after each frame.

At this point, we have particles successfully translating acceleration vectors into actual movement, but we haven't accounted for our constraints. To do this, we implement a form of strain-limitation as described by Thomas Jakobsen in "Advanced Character Physics"[1]. Instead of directly applying Hooke's Law using some spring constant k, this method instead calculates a direction vector along which we "correct" our particles. To do so, we want to keep track of a rest length between two particles, essentially an "equilibrium" distance which our particles strive to achieve. During the Verlet integration, particles move around, disrupting this equilibrium. We then loop over all springs and run this correction method, which looks like the following

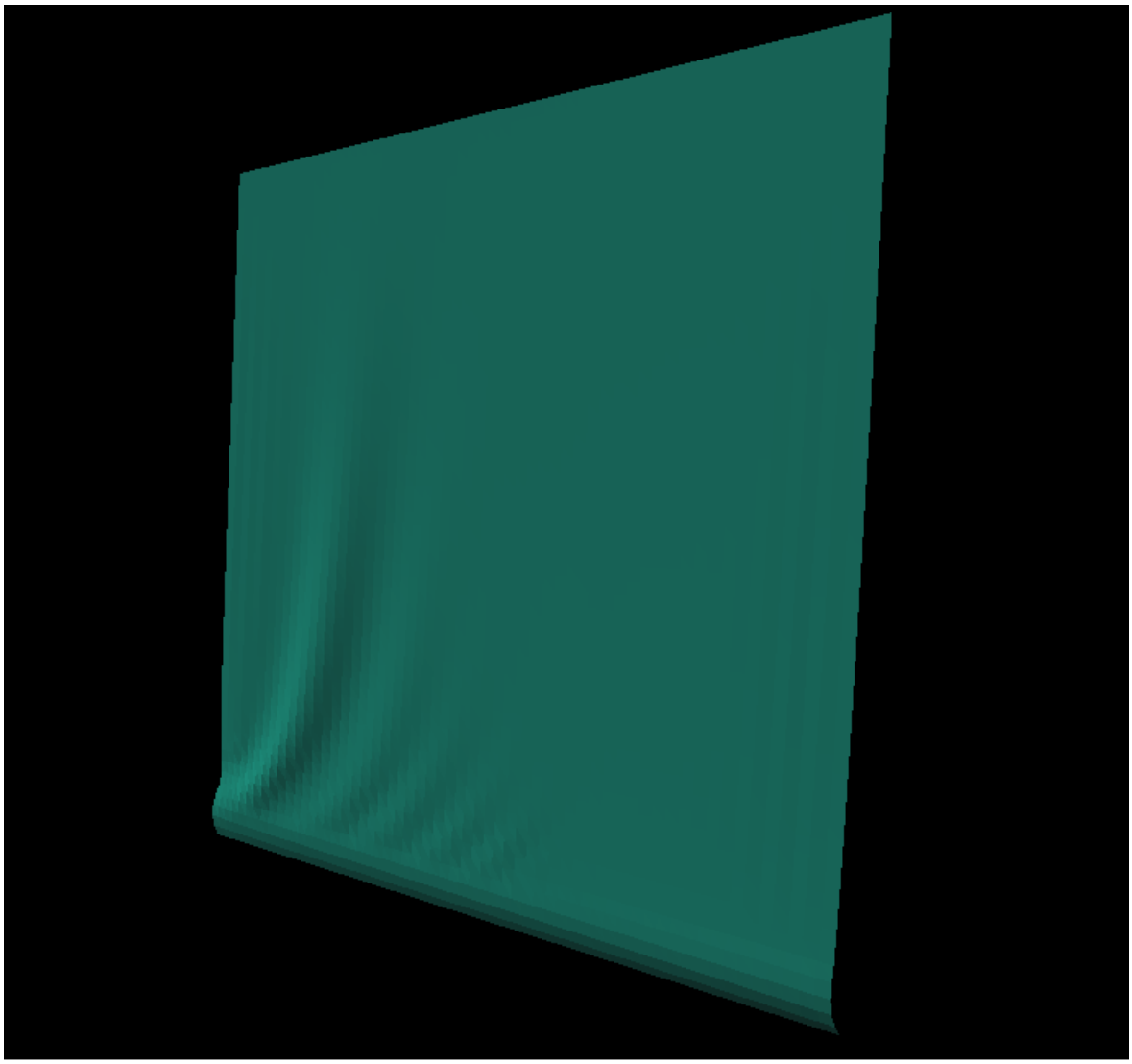
```
solveSpring() {
    vec3 delta = p2 - p1;
    float curr_dist = delta.length;
    vec3 correctionVec = delta * (1 - equilib_dist/curr_dist);
    correctionVec  /= 0.5;
    p1.updatePos(correctionVec);
    p2.updatePos(-correctionVec);
}
```

This moves each particle such that the difference is closer to its rest length, serving to constrain the particles in cloth form as Verlet integration provides movement to each particle.

4 Rendering

At this point, we have (what seems like) a working cloth – the particles stay attached and do not oscillate indefinitely. We need to render a whole cloth, so much like the teapot in Assignment 3, we have to somehow group and render each set of four particles as two triangles. We save some time by creating these Triangle objects in the same loop that creates constraints. After updating the position of the particles, we simply loop through all triangles (another vector in Cloth), rendering them as OpenGL polygons. We simply add some lights, and we have now have a flat-shaded, somewhat boxy looking cloth.

To implement smooth shading, we add a field to Particle called $accumNormal$, which serves as a vertex normal. When computing a face normal for a triangle, we can add this into the accumulated normal at each vertex, averaging to get a correct result.

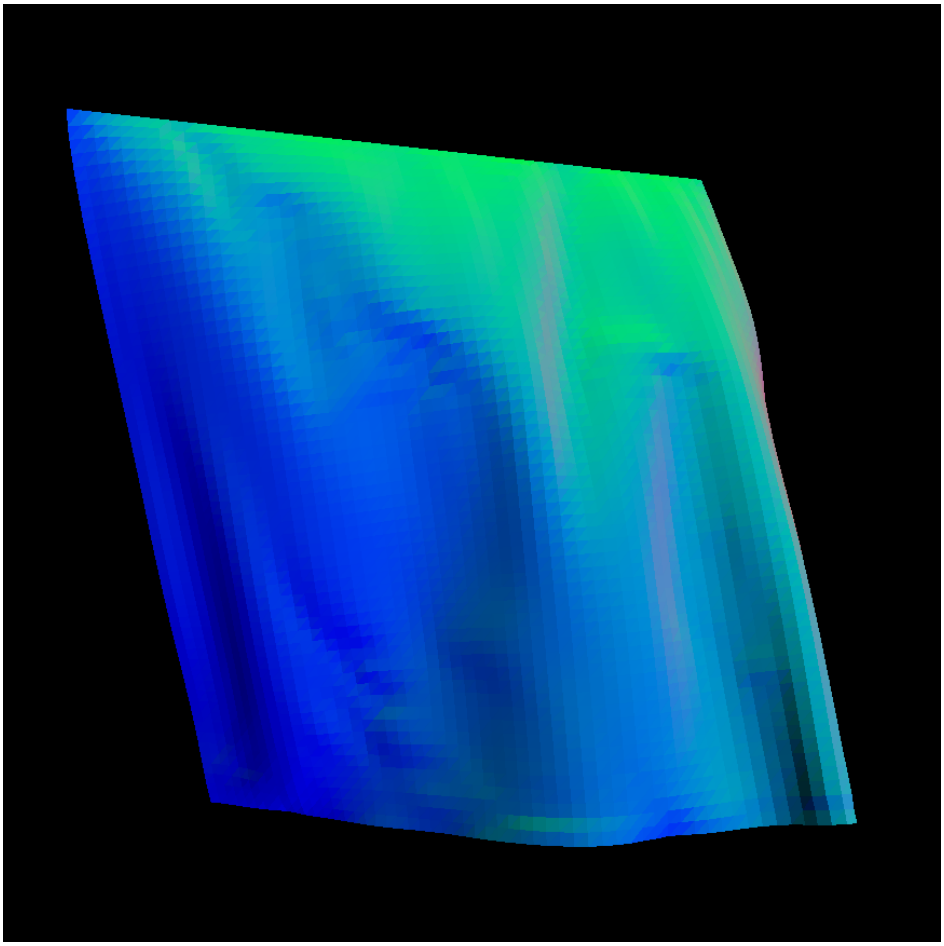


5 Adding Wind

This cloth, however, is boring. It only reacts to gravity, and with a fixed axis, sways only for a little while before settling down due to damping. We now introduce some external agents to interact with the cloth to spice it up a little bit.

```
void addWind(windVector) {
    for Triangle in Triangles {
        norm  = Triangle.getNormal();
        force = normalized(windVector) * dot(windVector, norm);
        for particle in Triangle {
            particle.addForce(force);
        }
    }
}
```

Each particle gets pushed around by wind, depending on the cross-sectional area of its triangle that lies in the plane perpindicular to the wind vector. Given the triangle framework we set up earlier, it is trivial to simply iterate through each triangle, adding these forces every frame.

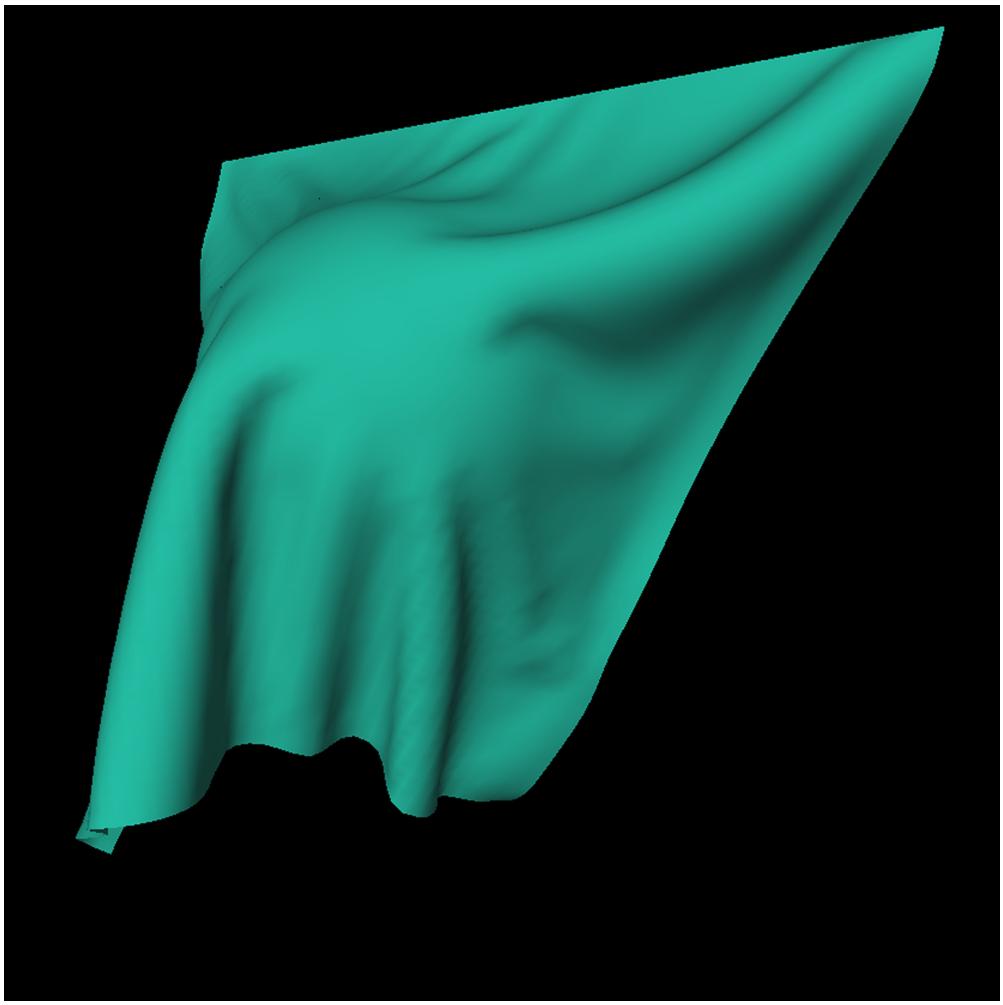


6 Sphere Collision

In addition to wind, interaction with solid objects is interesting to observe. We add in a Ball class, which merely represents a sphere. It holds only the position and radius of the sphere, whose posiiton is controlled entirely by the user.

```
void resolveCollisions() {
    for Particle in Particles {
        d = Particle.pos() - Ball.pos();
        if (Magnitude(d) <= Ball.radius + error) {
            Particle.updatePos(normalize(d) * (Ball.radius + error - Magnitude(d)));
        }
    }
}
```

After Verlet integration, we resolve the collisions using the method above. Basically, if a particle's tentative position is within the ball (with some buffer area to prevent artifacts), we simply force the particle to move outwards towards the edge of the sphere, in the same relative outward direction.



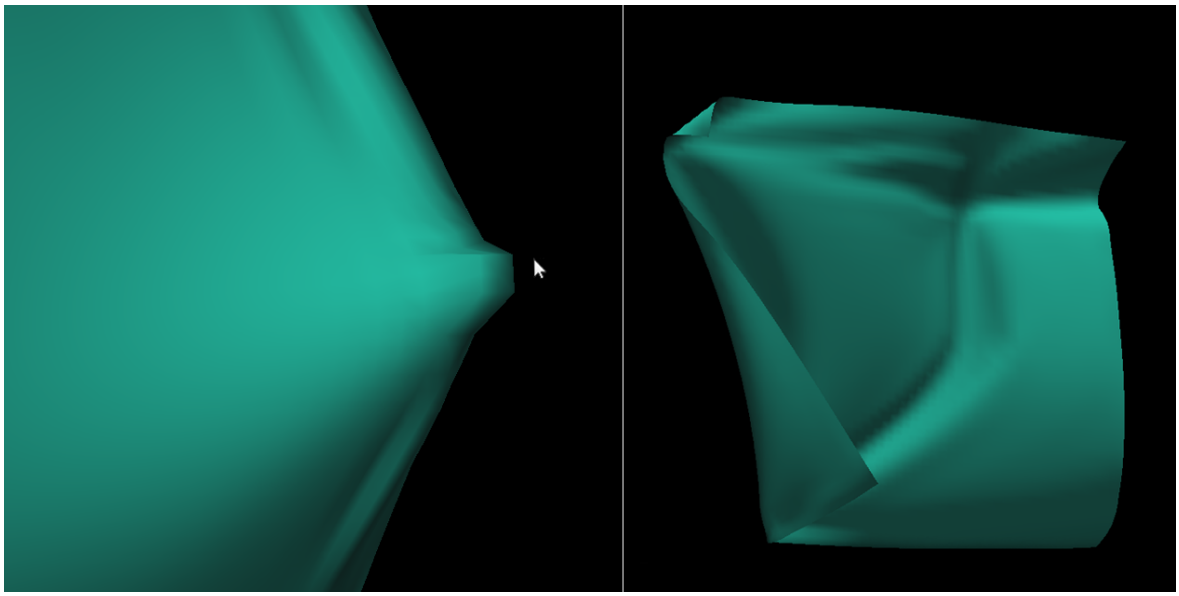
7 Mouse Interaction

Finally, the last feature to implement is some sort of mouse click-and-drag interface for the user. The first step is deciding which particle to grab when the user clicks his mouse. To do this, we need a transformation which can convert a 2D point on our screen to a 3D point in our world space. While X and Y are easy, how do we determine how deep "into" the scene we are clicking? Luckily, OpenGL provides a function called *glReadPixels*, which we use to grab a value from OpenGL's depth buffer. Basically, when we hover our mouse over a piece of our cloth, glReadPixels returns the Z value that the piece is rendered at.

Now, we have a mapping from our cursor position to a 3D world space point. The next step is finding the closest Particle to this point. We do this with our *dragCloth* method, which takes in the 3D world space coordinate, locates the nearest particle to this point, and tells our cloth that this particle is currently being dragged

```
dragCloth(vec3 clickPos) {
    for Particle in Particles {
        if (Particle closer than closestParticle) closestParticle = Particle;
    }
    draggedParticle = closestParticle;
}
```

At last, we can successfully click a position on a screen (which selects the closest particle), move our mouse (which moves the particle along some Δx, Δy, Δz), and release our mouse to release the particle from user control. To make the dragged portion of the cloth not appear as an ugly spike, we simply drag a particle and its neighbors. As a final feature, we add a keyboard shortcut 'p' to pin a dragged particle in place. This is simple, we already have the freezing functionality from fixing an axis of particles in place. Now we just set the dragged particle to frozen when 'p' is pressed. When the user isn't dragging a particle and presses 'p', all pinned particles are released and the cloth returns to normal.



8 Conclusion

This project was a rewarding experience and we are very happy with how it turned out. We successfully implemented a realistic cloth simulation, and even were able to add in some challenging features such as click-and-drag interaction. This project was an excellent way to combine most of the material from this course into a stimulating result. Given more time, our group would look into adding texture mapping and tearing, but overall we are very happy with the amount of features we implemented.

References

[1] Jakobsen, Thomas. "Advanced Character Physics." http://www.pagines.ma1.upc.edu/ susin/files/ AdvancedCharacterPhysics.pdf

[2] Mosegaard, Jasper. "Mosegaards Cloth Simulation Coding Tutorial." Computer Grpahics Lab. Alexandra Institute, 2009. Web. 17 Apr. 2013. http://cg.alexandra.dk/2009/06/02/mosegaards-cloth-simulation-coding-tutorial/