Leah Taurisano
VGP 256

Ragdoll and Cloth Physics Simulation

The goal of my physics simulation is to create an accurate representation of ragdoll physics using constraints to link one mass to another. These constraint links will keep the two masses locked to the specified distance from one another, meaning when one mass moves, the other must follow. When multiple masses with multiple constraints are linked to one another, this creates a cloth-like ragdoll visual.

The Masses

The most basic component in a ragdoll simulation is, of course, the mass itself. These masses have several components that enable their functionality within the simulation, for example a spatial component keeping track of their position and orientation, and a linear body component giving the masses independent velocity and mass.

My personal addition to the masses is a pinned component, which allows you to designate whether a mass is allowed to move in the world. If it is pinned, all physics calculations of the simulation are bypassed, allowing me to have stable points to better showcase the ragdoll physics.

The Constraints

The constraints are where the bulk of my implementation takes place. When a constraint is created, it is passed several pieces of information for its constructor, a link distance, a break threshold, and the two masses, which I will refer to as Mass A and Mass B, to be linked together.

Link Distance

The link distance is the specified distance that the two masses are forced to stay apart from one another. The first step is to calculate the distance between the two masses, by simply subtracting the position of Mass B from the position of Mass A. In a homogeneous coordinate system, subtracting two points, which would both have a ‘w’ value of 1, results in a ‘w’ value of 0, thus giving us a vector. As such, this returns their distance as a vector, pointing in the direction from Mass B to Mass A.

Now that we have their distance, we normalize it to simplify it to just the unit direction. This will be used to give each mass the direction it needs to move to move closer to its linked mass to obey the constraint.

Next, we calculate the midpoint between the two masses; this will provide a point for the two masses to effectively lock to and move towards. We do this by adding the two positions, and dividing it by two. This is done, once again, because of the homogeneous system. The two points will have a ‘w’ value of 1, adding them together gives us a value of two, and then dividing by two returns this 1, giving us a point as our result, and the center point between the two masses.

Finally, we can begin moving our masses closer, or father, from one another to make them obey their constraint. We do this by overriding each masses’ position with an entirely new position. We calculate this position by multiplying the link distance established during the creation of the constraint by the direction we calculated above, then multiplying this by half, because each mass only needs to consider half of the link distance in relation to the midpoint. We then add the midpoint at the very end, giving us a point in space exactly half the link distance away from the midpoint, in the direction of Mass B to Mass A. Since this direction only works for Mass A, we simply negate the vector part before adding the midpoint, flipping the direction and setting both objects at their approximate required distances from one another.

We do need to change the behavior slightly when calculating a constraint that is linked to a pinned mass, however, as the pinned mass cannot move at all, only moving the unpinned mass half the distance will no longer suffice. For this, we no longer multiply the directional link distance by half, and instead of adding a calculated midpoint, we add the position of the pinned mass. This ensures that the unpinned mass will move to exactly the link distance away from the pinned mass, while still respecting the direction between the two masses.

Break Threshold

The break threshold is, luckily, much more simple to handle. Before calculating the new positions for the constrained masses, we check the actual distance between the two points. While we calculated distance before, this, as mentioned, gave us a vector, so first we need to get the magnitude of this distance vector to have the scalar distance between the masses. Next, we multiply the link distance by the break threshold. If this value is larger than the distance between the masses, the entire constraint is destroyed, and so no further calculations will take place.

Relaxation

I have also implemented “relaxation” into my constraint simulation. Relaxation is the act of recalculating the constraint sim several times over to achieve a more accurate final result. This system works because when multiple masses are linked together through multiple constraints, sometimes applying the constraint to two linked masses could break the constraint to another mass linked to one of the first two. This problem would be small or unnoticeable in small links, or in an environment where the masses aren’t moving, but it becomes very apparent in the presence of forces, such as gravity. By applying the simulation multiple times, the masses will, eventually, be very close to the required positions in their constraints, preventing multiple linked constraints from too heavily affecting one another.

Simulated "Weight"

One, originally unintended, side effect of my method of constraint implementation, is that masses will "weigh down" other linked masses above them. This is essentially the problem that relaxation solves, and while increasing the amount of times the simulation "relaxes" the constraints removes this effect, there is only so many times you can calculate them before it begins to slow down the simulation. This did, however, have the very nice effect of giving weight to the linked masses, and is used in my demonstration of my breakable constraints. While originally not an intended feature, it added realism to the simulation, and overall improved it.

What I Did Not Add

Verlet Integration

In the original plan of my project, I had intended to implement Verlet Integration into the simulation. This was originally my intent because, from my research, Verlet would have simplified the math and handling required for the ragdoll simulation. After initial implementation, however, and with Euler Integration already implemented, I decided to move forward with Euler as my integration method. Implementing Verlet Integration would have involved removing velocity itself from the masses, and instead tracking their current and previous positions, and using this to calculate a stand-in for velocity. While it would have been interesting to also implement this system as an option, I am still very happy with how my simulation turned out.

Angle Constraints and Actual Ragdolls

While it may seem odd to not have physically assembled ragdolls in a ragdoll simulation, I came to understand that simulating ragdolls and cloth are effectively identical. Creating a ragdoll would be as simple as creating masses and constraints in the appropriate shape and distances to make something resembling a person. This ragdoll would, however, have one fatal flaw - the only reason I did not end up making the jump to a full ragdoll - a lack of angle constraints.

I unfortunately did not have time to implement angle constraints. I believe this would have been done by calculating the angle between the position of, say, Mass B, in relation to the angle of Mass A. Mass A would effectively create a straight line down to Mass B, and if Mass B rotated outside of an allowed angular range in relation to this line, the constraint would keep the mass inside, and prevent further movement outside of that range. If I have more time in the future to return to this project, this would likely be the first thing I try to implement.