Lingtao Chen 305547772
Yili Liu 205376049
Michael Mooc 905095989

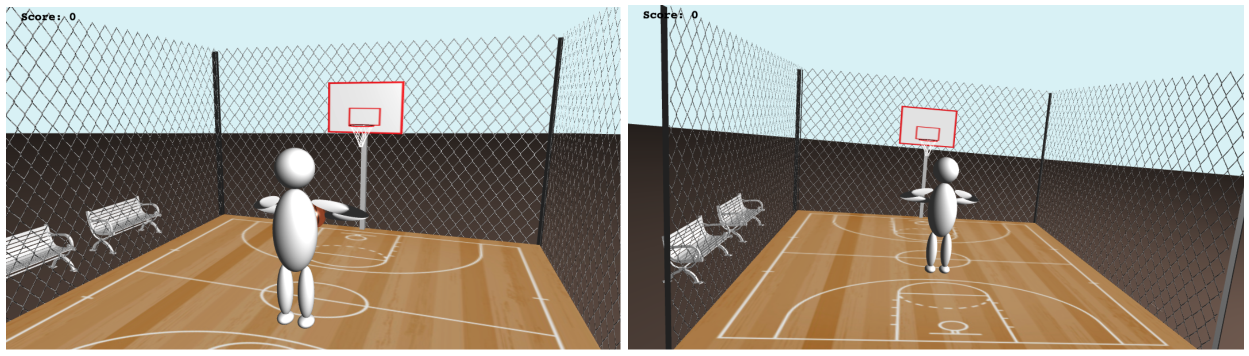
CS 174C Final Report: Basketball Simulation

1 Introduction

Our project was focused around making a simulation where a player can shoot a ball into a hoop from a variety of positions on the court. We were inspired by the fact that we could apply so many different animation techniques to the game of basketball. The aim of the simulation is to shoot the ball into the basketball hoop as many times as possible. The animation starts with the player holding the ball in their hands. The player sees the court in a third person perspective and can manipulate the curvature of the ball's arc and where it is thrown through user interface buttons: angle up, down, left, right, more and less power. In order to help guide the player, a spline showing the firing arc is displayed on the court as well before the ball is released.

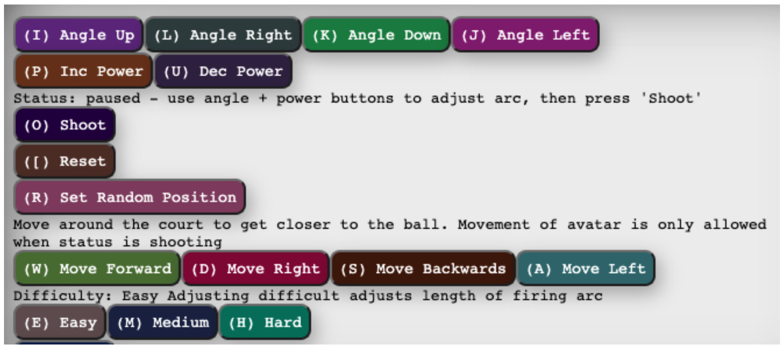
When the ball is thrown, the firing arc disappears, and the player's arms move along with the ball and back towards the resting position with their hands on both sides. Additionally, collision detection is used to determine if the ball made it through the hoop. If the ball successfully goes through the hoop, the net under the hoop, made from a mass spring damper system, will animate accordingly and the score label on the upper left corner will update as well.

2 Scene Design and User Control



Figures 1a), 1b). Scene Design

As seen in *Figure 1a* and *Figure 1b*, our scene consists of a basketball court with a simple basketball hoop at the back center of the court. The court is surrounded by a wired fence and two benches for decor. Outside of the court, there is a simple ground and sky. The top left corner also displays a score, showing how many shots have been made.



```
Status: paused - use angle + power buttons to adjust arc, then press 'Shoot'
Move around the court to get closer to the ball. Movement of avatar is only allowed
when status is shooting
Difficulty: Easy Adjusting difficult adjusts length of firing arc
```

Figure 2. Control Panel

The user control is done through the control panel. To change the firing arc of the ball, the user can select from the angle up/right/down/left buttons and increase/decrease power. The shoot button shoots the ball. It has two states - paused and shooting. In the paused state, the user can adjust the firing arc. In the shooting state, the ball is shot and the user can move forward/right/left/back to get to the ball before shooting again.

The Reset button resets the ball and the player to the center of the court. The Set Random Position button resets the ball and the player to a random spot on the court. There are also three difficulty buttons - easy, medium, and hard which correspond to how much of the firing arc is displayed/how long the firing arc is.

2 Algorithms

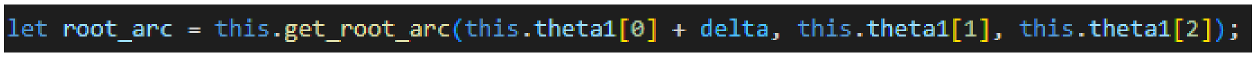
In total, we incorporated five algorithms from the course in our simulation: inverse kinematics, symplectic euler integration, hermite splines, mass spring damper system, and collision detection and resolution. Each algorithm will be discussed in the following sections.

2.1 Inverse Kinematics

Inverse Kinematics was used to move the player's arms along with the ball's motion throughout the simulation. We used the Assignment 2 code as the template, which was based on TA's code, and added the left side IK.

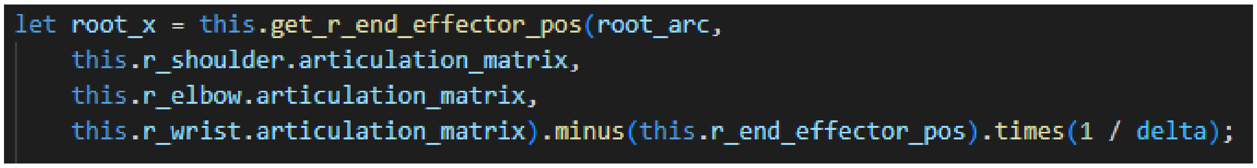
For one side of the human body, we focused on the hand movement, so there were 10 degrees of freedom(DOF). However, we didn't use the translational DOF at the root, because the motion was weird. We simply set the corresponding Jacobian column to be 0.

The core was to calculate the Jacobian matrix. We used the numerical approach. The Jacobian matrix should be a 3x10 matrix. For each DOF, we made a very small change and left the others unchanged.



```
let root_arc = this.get_root_arc(this.theta1[0] + delta, this.theta1[1], this.theta1[2]);
```

Using the root as an example, this results in a new articulation matrix. We then can use this to calculate the new position of the end effector.



```
let root_x = this.get_r_end_effector_pos(root_arc,
    this.r_shoulder.articulation_matrix,
    this.r_elbow.articulation_matrix,
    this.r_wrist.articulation_matrix).minus(this.r_end_effector_pos).times(1 / delta);
```

We then can get the difference between the new and old positions of the end effector. The difference being divided by the small change, which is delta, would yield a vector.
This vector indicates how the change in this specific DOF would affect the change in position of the end effector.

Repeat for the rest of the DOF to get a 3x10 Jacobian matrix.
Find the pseudoinverse of the Jacobian.
Given a change in position, we can use the pseudoinverse to find the corresponding change in all DOF.
Update all DOF and update all articulation matrices.

All the above implementations were about the right side of the body. To have the same effect on the left side. We copied the entire code and made a left-side version. They had exactly the same implementations.

There were 3 states, idle, shooting, and end states. The position of the ball was passed in as the goal position of the inverse kinematics, so both hands would move along with the ball in the first two states. When the ball was thrown and unreachable, it was in the end state. Both hands would move to the final positions, which were the left- and right-side of the character, which were also the goal positions of the inverse kinematics.

2.2 Symplectic Euler Integration

Symplectic Euler Integration was used to calculate the general movement of the ball and the net. The following formulas were used:

$$v(t + \Delta t) = v(t) + \Delta t\ f(x(t), v(t), t)/m$$
$$x(t + \Delta t) = x(t) + \Delta t\ v(t + \Delta t)$$

Through a series of external force computations, the ball and the particles of the net had the latest external force exerted on them for each update. Given the force and the mass, their acceleration could be easily calculated using Newton's law. Based on the timestep, the new velocity could be calculated, and further the new position of the ball and the particles could be calculated.

The ball had *external force* as the member variable. In the idle state, the force was 0, and the players could exert force on the ball by adjusting the angle and power through user control. Once the players shot the ball, other forces would be exerted, such as the gravity, friction, and forces due to collisions with the walls, the ground, and the hoop.

We used the formulae provided in the lecture slides for the friction model and wall and ground collision models.

Viscous (linear) friction model: $\mathbf{f}_v = -\mu_k \|\mathbf{f}_\mathrm{n}\| \mathbf{v}_\mathrm{t}$

(friction formula)

$$\mathbf{f}_n = k_s \left((\mathrm{P_g} - \mathbf{x}(t)) \cdot \hat{\mathbf{n}} \right) \hat{\mathbf{n}} - k_d (\mathbf{v}(t) \cdot \hat{\mathbf{n}})\, \hat{\mathbf{n}}$$

$$\mathbf{f}_n \cdot \hat{\mathbf{n}} > 0$$

(ground and wall normal force formula)

2.3 Hermite Splines

Hermite Splines were used to display the firing arc of the ball in a thin blue line. Every time the user clicks an angle up/down/left/right or increase/decrease power button, a new hermite spline is generated based on the motion of the ball. The hermite spline's points and tangents correspond to the position and velocities at different points in time as seen below in *Figure 5*.

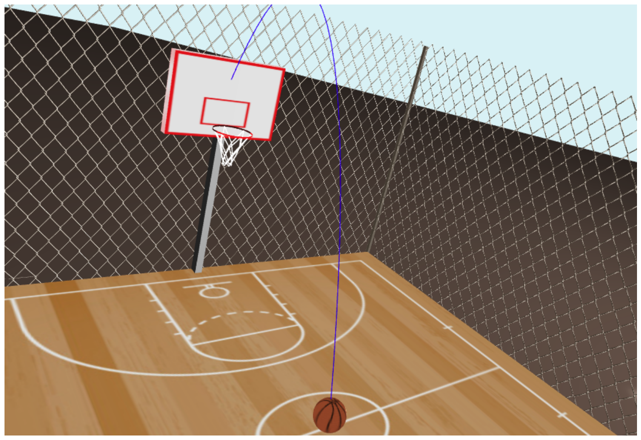


Figure 5. Hermite Spline showing firing arc

The spline's length is dependent on the difficulty the user selects. With the difficulty set to 'Easy', a spline curve is displayed until it hits a wall or the ground so that the player knows exactly where the ball is heading. To do this, we used a while loop to keep generating positions and velocities at different time steps until we detected that a position touched a wall or the ground. In the other difficulty levels 'Medium' and 'Hard', the while loop has an extra condition in which if the spline reaches a certain length, we break out of the while loop so that the spline is not as long.

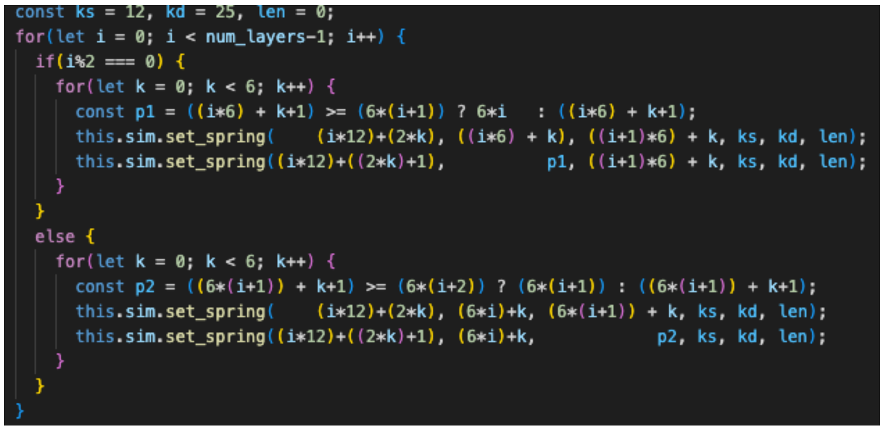
As for the calculation of position and velocity used in the splines, we followed the exact same formulas as we use for the actual ball movement. These formulas are as follows:

$$F\_e = g*mass + Force$$
$$a(t) = F\_e / m$$
$$v(t + \Delta t) = v(t) + \Delta t*a(t)$$
$$x(t + \Delta t) = x(t) + \Delta t\ v(t + \Delta t)$$

Here, F_e is the external force on the ball which includes gravitational force and external force from updating the angles and power on the ball. a(t) is the acceleration and using F_e and a(t), we get the velocity and position that we use as points and tangent on the spline. In order to make it so that the spline is not full of too many points, we only add the position and velocity every 100 iterations in relation to Δt.

2.4 Mass Spring Damper System

The mass spring damper system was used to construct the net. The net is made out of five layers of particles in the form of a circle. Each particle in the second to fifth layers is connected to two of the particles in the layer above it by springs which allows the net to hold its form.



```
const ks = 12, kd = 25, len = 0;
for(let i = 0; i < num_layers-1; i++) {
  if(i%2 === 0) {
    for(let k = 0; k < 6; k++) {
      const p1 = ((i*6) + k+1) >= (6*(i+1)) ? 6*i   : ((i*6) + k+1);
      this.sim.set_spring(    (i*12)+(2*k), ((i*6) + k), ((i+1)*6) + k, ks, kd, len);
      this.sim.set_spring((i*12)+((2*k)+1),          p1, ((i+1)*6) + k, ks, kd, len);
    }
  }
  else {
    for(let k = 0; k < 6; k++) {
      const p2 = ((6*(i+1)) + k+1) >= (6*(i+2)) ? (6*(i+1)) : ((6*(i+1)) + k+1);
      this.sim.set_spring(    (i*12)+(2*k), (6*i)+k, (6*(i+1)) + k, ks, kd, len);
      this.sim.set_spring((i*12)+((2*k)+1), (6*i)+k,            p2, ks, kd, len);
    }
  }
}
```

Figure 6. Code for mass spring damper system initialization of net

To determine which particles in the above layer a particle is bound to, we used the code shown in *Figure 6* where the first and second argument of this.sim.set_spring() are the particles that connect the spring.

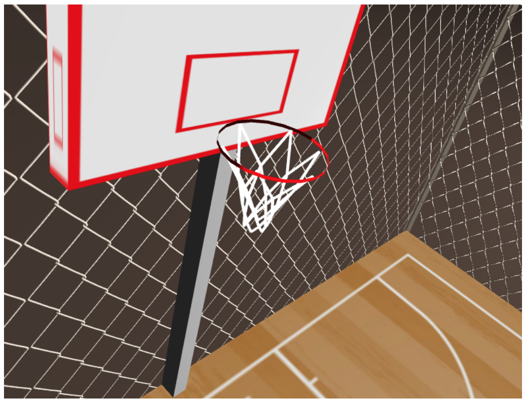
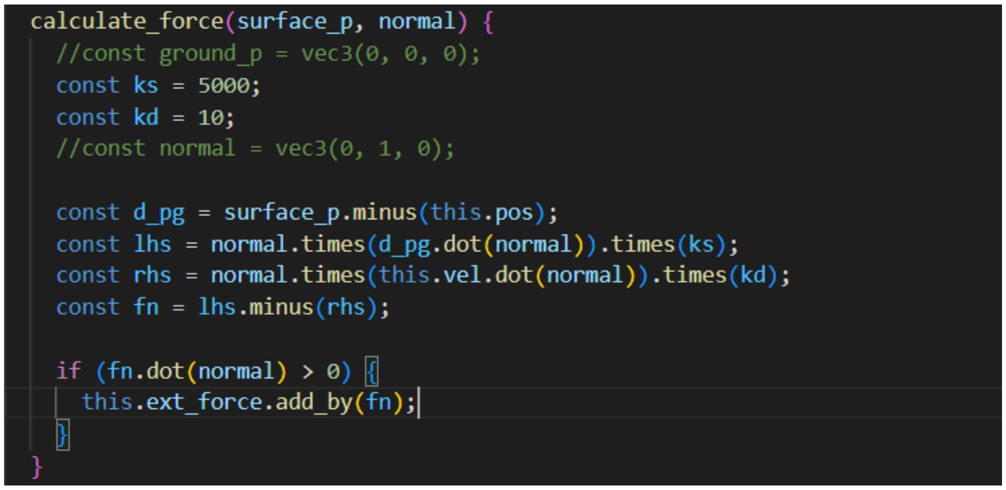


Figure 7. Close up of net shape

Figure 7 shows the result of the code above and the mass spring damper system. The formulas for the spring force we used were the ones given in the lecture slides. Specifically, **f** = ks(d - l)**dir** + kd(**v** · **dir**)**dir**. Where dir is the direction, v is the velocity difference between particles i and j, ks is the spring constant, kd is the damper constant, d is the distance between the particles, and l is the length of the spring. Additionally, symplectic Euler integration was used to update the particle motion as discussed in the above sections. Further net movement during collision detection will be discussed in the following section.

2.5 Collision Detection and Resolution

Collisions with static surfaces such as the floor and walls are handled by member functions within the ball object. These functions check if the ball makes a collision with a wall by taking the distance between the current position of the ball and a surface, and multiplying it with the surface normal and the ball's velocity. The resulting force calculation is applied to the ball's ext_force values. This process is repeatedly called for every surface.



```
calculate_force(surface_p, normal) {
  //const ground_p = vec3(0, 0, 0);
  const ks = 5000;
  const kd = 10;
  //const normal = vec3(0, 1, 0);

  const d_pg = surface_p.minus(this.pos);
  const lhs = normal.times(d_pg.dot(normal)).times(ks);
  const rhs = normal.times(this.vel.dot(normal)).times(kd);
  const fn = lhs.minus(rhs);

  if (fn.dot(normal) > 0) {
    this.ext_force.add_by(fn);
  }
}
```

For collisions with the net, the distance between every particle with respect to the ball is checked. If any particle of the net is found to be within the radius of the ball then an elastic collision would be calculated, along with pushing the particle out of the radius. Elastic collision formula: $m_1u_1 + m_2u_2 = m_1v_1 + m_2v_2$, where ‘u’ represents the initial velocity and ‘v’ is the final velocity. Preventing the particle from staying within the radius is done by creating a unit vector between the particle and the ball's center. Following the unit vector’s direction, place the particle in a position that is outside of the radius.

3 Challenges

One of the challenges we faced was making net movement look natural on collision with the ball. To actually calculate an elastic collision requires context of the aftermath of a collision. Since we can only provide the initial velocities, it is not possible to calculate the correct resulting velocity without creating a complex physics engine. Therefore, our solution is to estimate what the resulting velocities would be by splitting the total momentum of a collision between the particle and ball. The ball would keep a large majority of the momentum, while the particle is given a little. The value of the split was manually calculated until a happy median was found.

Additionally, it was hard to make the human look natural with inverse kinematics. The hands didn’t always go in the way we wanted. The joints were much affected by the current articulation matrices. We then decided to reset the articulation matrices to the initial state when the character was moved, such as reset was called and random position was called.

4 Division of Work

We divided the work mainly by the algorithms.

Lingtao worked on the Inverse Kinematics, the human character, general physics and movement of the basketball, and collision detection between the ball and the walls/ground.

Yili worked on the Hermite Splines/firing arc, the general build of the net, collision detection for the main hoop body and rim, and building the scene (textures, fence, added decor, hoop, etc.)

Michael worked on the scoring functionality, buttons/control panel, and collision detection and movement of the net.

5 Conclusion

In conclusion, we were able to take multiple animation algorithms learned in class to make a basketball simulation. These algorithms included inverse kinematics, collision detection and resolution, symplectic euler integration, hermite splines, and the mass spring damper system. In the future, some ways that we could expand this project are to use more advanced algorithms to model more natural human motion, beautify the scene more by adding elements such as clouds made from particles, and introduce wind as an element of the game that affects the trajectory of the ball to make the game more interesting.