Basketball Shooting Simulation

Kelly Liu and Aaron Nojima

I. Motivation

Basketball is a well-established sport in which the main objective is to shoot a ball into the rim. There are many other rules and much more complex strategies than just this but shooting is the fundamental skill of basketball. Although most athletes depend on practice and muscle memory, understanding the physics behind shooting is very important. For example, understanding how the ball will interact with the rim or backboard or figuring out what angle and speed one should shoot the ball given a specific position on the court will help anyone learn how to shoot with greater accuracy. Thus, we have created a basketball shooting simulation where the user can control the direction and velocity of the ball in order to practice shooting in a virtual environment with some realistic physical behavior. This project was inspired by basketball games such as the NBA 2K series. These games have great graphic implementation and easy-to-use controls for playing a basketball game. However, we wanted a lightweight, easy-to-access simulation that requires users to think and experience the physics behind shooting a basketball as opposed to simply pushing a designated "shooting" button.

II. Background

This project does not really build off previous work or research, but we utilized various existing graphics libraries, resources, and information found available on the Internet. Since we wanted to make an easy-to-access simulation, we wanted to to create a simulation that could be rendered on a common platform, the Internet. Thus we decided to implement our simulation using JavaScript. There is one primary graphics library available for programmers called Three.js [1]. Three.js already implements a significant portion of rendering functionality, but by itself, does not support physics or game engines. Most of our work involved creating a game engine specific to the requirements for our basketball simulation.

In addition, since Three.js already implements texture mapping for various geometrical solids (such as spheres, planes, and boxes), we decided to simply find existing texture mappings online for the basketball and court. We consulted www.robinwood.com [2] for the basketball texture map and www.cupcakerie.au [3] for the basketball court texture map. For the basketball rim we created our own texture map (black rectangle on white) and for the basketball rim and net we simply made solid colors.

In order to give the realistic appearance of relative size, we wanted to use actual dimensions (converted to a chosen value to pixel ratio). For the dimensions of the basketball, court, rim, and backboard, we consulted www.nba.com [4] for official lengths and measurements as well as for the mass of a basketball.

We consulted www.wolframalpha.com for the cartesian equation of a torus [5] and for the quartic equation [6] used primarily in rim geometry and collision handling.

III. Approach

The work for our simulation can be divided into four main phases: the design of the 3D objects, implementing projectile motion for the basketball, creating a stable and responsive mass-spring system for the net, and handling collisions for all possible objects interactions. We also added a control system for ease-of-use and debugging purposes.

A. Design of 3D Objects

We either obtained or produced texture maps for the following objects in our scene: the basketball, the backboard, the court. The other objects that required coloring in our scene is the rim and the net, for which we simply assigned a diffuse color of an appropriately matching orange and white, respectively. For the basketball and court, we used pre-generated 2D-texture image maps and utilized the texture-loading functionality provided by Three.js. The backboard, being a box, required six 2D-texture images that we designed by hand (a white image with a black box in the middle or just a blank white image for the sides, top, bottom, and back of the backboard).We also used Three.js functionality to add reflectivity and shading to our objects.

B. Projectile Motion of Basketball

We implemented Euler time-stepping in our main render function for the basketball object. This uses current state (position and velocity) of the basketball. The time stepper evaluates the derivative of these properties for the basketball by taking the current velocity of the basketball, all external forces applied to the basketball (primarily gravity for projectile motion), as well as the timestep. Given the derivatives and time-step we calculated the next position and velocity of the basketball. For angular velocity, we did not account for friction, however we observed that the basketball spun in a certain direction (along an axis) proportional to its x- and z-velocity (not upward or downward speed). At every time-step, we calculated the next angular velocity based on these velocity component values.

C. Net Mass-Spring System

We modeled the basketball net as a mass-spring system. Each “knot” (rendered as a spherical geometry) in the net represents a particle in our system model and each “line” connecting any two “knots” represents a spring. The top layer (the layer of knots attached directly to the rim) will be fixed in place (ignoring velocity and acceleration in the time-stepper render method). Since the lines between the top layer and the one below it are noticeably longer than other lines we made the relaxed distance higher than the other springs. Unlike the cloth system implemented in the third problem set, the net is not a continuous sheet--the springs are represented by real tangible strings. Thus we only added structural springs in our system. Each particle’s state is calculated in a similar manner to that of the basketball: compute derivatives (this time including all the forces exerted from the attached structural springs) and find the next state.

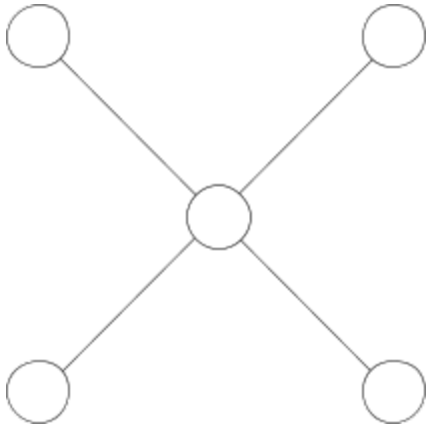

Figure 3: Net - knots as particles and lines as springs

D. Collision Handling

In our simulation, there are two sets of objects: dynamic (can move e.g.: basketball, net particle) and static (are fixed e.g.: court, backboard, and rim). We currently handle collisions between the basketball and all static objects as well as between the basketball and the net. In order to handle collisions, we provide the two objects of interest (of which we know one will always be the basketball). We expect to receive a boolean claiming whether or not the two objects overlap or intersect each other and if so any other information relevant to fixing the collision.

boolean handleCollision(3DObject basketball, 3DObject obj2)

1. Add padding of thickness = basketball.radius to obj2 ⇒ pObj2
2. If basketball.center is within pObj2:
    a. Create ray: origin = basketball.center, direction = -1*basketball.velocity
    b. See where the ray LAST intersects pObj2 (max t value) ⇒ fixPoint
    c. Set basketball.center tto fixPoint
    d. return true
3. Else: return false

First, by padding the second object with the radius of the basketball, we essentially reduce the problem from two objects to one object and a point. If the center of the basketball lies within the padded object, this is the same as saying that some point on the surface of the basketball would be lie within the original object. The padded object depends on the original geometry. For example, a plane becomes a box, a sphere becomes a larger sphere, a box becomes a larger box with rounded corners and edges, and a torus becomes a larger torus.

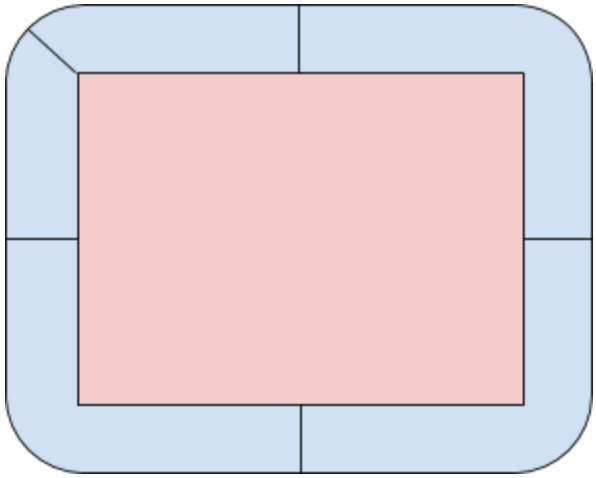

Figure 1: Padded rectangle for collision detection

Second, for collision position-fixing, we must find the point on the basketball that would have first come into contact with the other object and set the center of the basketball to that point position. Again let us use our padded object and the center of the basketball. Let C1 be the current center of the basketball and let C0 be the center of the basketball when the ball first came into contact with the other object. We know that the ray generated from C0 to C1 has the same direction as the velocity since it's the direction from where the basketball came. We also know that C0 must lie on the surface of the padded object since any point on the surface represents a basketball's center position that just came into contact with the other object. Thus the problem is reduced to finding the intersection of a ray and the padded object. This again depends on the other object's geometry. For example, ray-sphere involves the quadratic equation ray-torus involves the quartic equation.

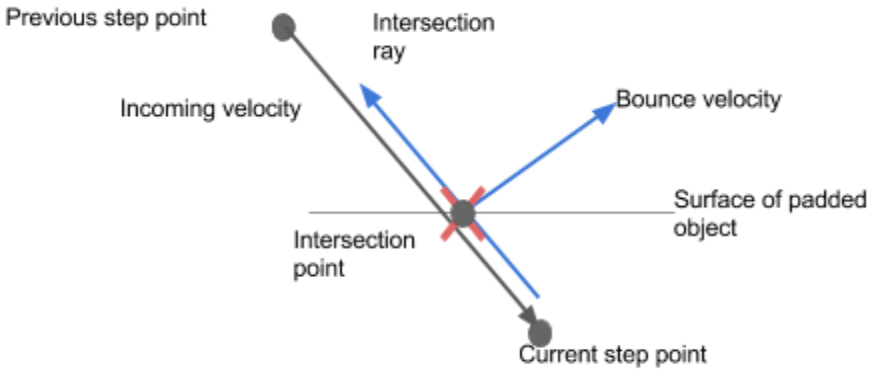

Figure 2: Collision fixing using position, velocity, and padded object

E. Controls

In order to make our simulation easy-to-use, we added a control menu. This menu allows users to play and pause the simulation at demand, always have the camera focus on the basketball, set the position or velocity, or reset all values back to zero. To give the user a better sense of "aim" when attempting to shoot the basketball, we display a velocity vector (informs the user of trajectory and relative speed) at which ball will be shot. All of these controls were implemented using JavaScript event listeners and Three.js methods.

IV. Results

Our simulation is fully-running as has the following features:

- Realistic Visuals
- Realistic Motion
- Realistic Collisions
- Easy-to-Use Controls

The following video will demonstrate the aforementioned features:
https://www.youtube.com/watch?v=rI3kop8xqP8

However, there are still various issues with our simulation. First, the basketball sometimes passes through the lines of the net. The issue here is due to the fact that the lines are not 3D Objects that can interact with other objects, but are merely lines rendered by the scene. Secondly, there are some instances when the simulation will display "jumpy" behavior. This is probably dependent on the time-step and comes down to a tradeoff between speed and collision accuracy. Some concrete examples of time-step issues include incorrect collision position-fixing, an unstable mass-spring system for the net, and non-converging bouncing on the court.

V. Conclusion

We were able to successfully create a basketball shooting simulation that allows users to test out different initial positions and velocities for shooting. We ended up utilizing a lot of the scene rendering and 3D mesh object functionality provided by the Three.js graphics library and thus focused on creating a realistic physics engine. The novelty of this simulation lies in our collision detection algorithms which do a near-perfect job of fixing the basketball position. That being said there is still work to be done regarding the time-step size issues which mainly comes down to the question of frame rate vs. accuracy. But overall, the simulation satisfies the needs of a physics-loving basketball fan.

References

[1] http://threejs.org
[2] http://www.robinwood.com/Catalog/FreeStuff/Textures/TextureDownloads/Balls/BasketballColor.jpg
[3] http://www.cupcakerie.com.au/assets/full/A4_BasketballCourt.png
[4] http://www.nba.com/analysis/rules_1.html?nav=ArticleList
[5] http://www.wolframalpha.com/input/?i=torus+equation
[6] http://mathworld.wolfram.com/QuarticEquation.html