FEBRUARY 22, 2021

EAGLE SCOUT PROJECT

RAFTSIM DESIGN DOCUMENT

Ishan Madan
EAGLE SCOUT CANDIDATE

TABLE OF CONTENTS

1 INTRODUCTION

1.1 ABOUT RAFTSIM

Resource Area for Teaching (RAFT) is a nonprofit organization that ships STEAM kits to thousands of classrooms around the US to supplement educational instruction in those fields. RAFTsim is an open-source project designed to either complement or replace these physical kits, which is especially useful during virtual classes. It is hosted through GitHub Pages and uses 3D models (created with CAD software) animated by the JavaScript library three.js. The simulations work with both desktop and mobile browsers, with added support as iOS web apps.

1.2 CONTRIBUTORS

Andrew Crabtree

Ani Aggarwal

Chris Park

Jonathan Joseph

Kedaar Chakankar

Sainirnay Mantrala

Seth Hondl

Sriya Kantipudi

Vishnu Velayuthan

Special thanks to Eric Welker, Director of Educational Content at RAFT

1.3 PURPOSE

This design document outlines the code and information utilized by the RAFTsim Kit Simulations project. It reviews the functionality of each of the 10 initial simulations presented to RAFT and created by the Contributors in section 1.2. Finally, it covers the process of creating a new simulation from another RAFT kit. More detailed instructions on creating a simulation can be found in the Maintenance Document or on the

RAFTsim GitHub repository at https://github.com/raftsim/raftsim.github.io#instructions-on-creating-new-simulations.

2 CAD

In the process of creating the simulations, two different CAD applications were used. The following simulations use models created in Autodesk Fusion 360:

- Hovercraft
- Leonardo's Bridge
- Tongue Depressor Harmonica
- Zippy Catapult

The following simulations use models created in OnShape:

- Black and White Makes Color
- Floating Compass
- Gravity Defying Frog
- Roller Racer
- Scallop String Art
- Waterbeads

This split in CAD software is due to the pre-existing experience of the Contributors with either application, however, there is no practical difference in the simulations between the models from each application. The only difference arises when importing the models into code, at which point there is a different template folder of code for the .obj files exported by Fusion 360 and the .stl files exported by OnShape.

3 CODE

Each simulation is built by modifying a template with the basic necessary structure. The code is entirely frontend (HTML, CSS, and JavaScript), and utilizes a third-party graphics library, three.js, to import, display, and manipulate models. Three.js, in turn, uses WebGL to carry out those tasks. Inputs are received using the HTML `<form>` and `<input>` elements, and the inputted information is read by JS.

Each simulation has a single root-level folder, containing index.html, stylesheet.css, and app.js files, along with an objects folder containing all the .obj or .stl model files used by the simulation. There are some exceptions to this format, like a separate simulation-level sound folder for sound files and .png images in the objects folder.

Below, see the template-stl version of each of the three files, with comments explaining the code.

3.1 INDEX.HTML

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Template | raft Kit Simulations</title> <!-- defines the title
of a given simulation (appears in the browser tab, history, etc) -->
    <link rel="stylesheet" type="text/css" href="stylesheet.css"> <!--
connects to css stylesheet file -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, user-scalable=no,
minimum-scale=1.0, maximum-scale=1.0"> <!-- manually sets full-screen on
mobile devices -->
    <link rel="shortcut icon" href="/storage/favicon.ico" type="image/x-
icon"> <!-- sets tab/bookmark icon to raft's hexagon "r" icon -->
</head>

<body>
    <script type="module" src="app.js"></script> <!-- connects to
javascript file -->
    <div id="form-container">
        <form id="form"> <!-- marks beginning of HTML form => gets user
input -->
            <a href="/">raftsim Home</a><br> <!-- link to homepage -->
            Example Input 1: <input type="number" value=0 id="input1"><br>
<!-- example input => takes a number with default value of 0 -->
            Example Input 2: <input type="number" value=0 id="input2"><br>
            <div id="submit-div">
                <button type="button" id="submit">Start</button> <!-- start
button -->
            </div>

            <br>
            <br>

            <span id="output-text">Example Output: <span
id="output">123.45</span></span> <!-- example output text, hidden by
default => number value is a separate span element to allow for editing -->
        </form>
    </div>
</body>

</html>
```

3.2 STYLESHEET.CSS

```
@import url('https://fonts.googleapis.com/css2?
family=Roboto+Condensed:wght@300&display=swap'); /* imports the Roboto
Condensed font from Google Fonts */

body { /* sets page background to black & font to Roboto Condensed */
    margin: 0;
    font-family: 'Roboto Condensed', sans-serif;
    background-color: black;
}

canvas { /* fixes graphics canvas in place as a "newline" element */
    display: block;
}

#form-container { /* sets margin around input form, positions form relative
to its first positioned element */
    margin: 20px 0px 0px 20px;
    position: absolute;
}

form { /* sets text details of input form, directs text/inputs to align
with right side */
    font-size: 20px;
    color: white;
    float: right;
    text-align: right;
}

input { /* sets text details inside input box */
```

```
    font-size: 18px;
    margin-left: 10px;
    height: 20px;
    width: 100px;
}

#output { /* sets output display style, size, & position */
    display: inline-block;
    margin-left: 10px;
    height: 20px;
    width: 105px;
    text-align: left;
}

#output-text { /* sets text details of output & hides it by default */
    font-size: 22px;
    visibility: hidden;
}

#submit { /* customizes green start button */
    font-family: 'Roboto Condensed', sans-serif;
    background-color: green;
    border-width: 1px;
    border-radius: 5px;
    border-color: transparent;
    color: white;
    padding: 3px 8px;
    width: auto;
    height: auto;
    text-align: center;
    text-decoration: none;
    font-size: 15px;
    margin: 8px 0px 0px 0px;
    cursor: pointer;
}

#submit-div { /* sets position & size of start button */
    width: 106px;
    height: auto;
    float: right;
    text-align: center;
}

a { /* sets text details of homepage link */
    color: white;
    float: right;
    text-align: center;
    width: 100%;
}
```

3.3 APP.JS

```
import * as THREE from 'https://unpkg.com/three/build/three.module.js'; //
import three.js
import { STLLoader } from 'https://unpkg.com/three/examples/jsm/loaders/
STLLoader.js'; // STL loader allows for importing .stl files as 3D objects
import { OrbitControls } from 'https://unpkg.com/three/examples/jsm/
controls/OrbitControls.js'; // orbit controls allow for dragging &
scrolling to move camera around a point (in 3D space)

var container; // html div container object
var camera, scene, renderer, controls; // camera - user's viewpoint, scene
- contains all other 3D objects & camera, renderer - updates the user's
view w/ new data about scene, controls - OrbitControls object
var assembly; // template 3D object variable

let targetPos = new THREE.Vector3(0, 0, 0); // sets the central position
for the camera to orbit around

var input1, input2; // variables to contain input data from user

let input1Min = 0; // min & max values of each input
let input1Max = 99;

let input2Min = 100;
let input2Max = 199;

let input1Input = document.getElementById("input1"); // objects
corresponding to html input tags
let input2Input = document.getElementById("input2");

var output = ""; // output starts as blank by default

init();
animate();

function init() {

    container = document.createElement('div'); // create container div
element to contain graphic elements
    container.id = "container";
    document.body.appendChild(container);

    var submitInputsButton = document.getElementById("submit"); // access
start button
    submitInputsButton.onclick = submitInputs; // run submitInputs() method
on button press

    camera = new THREE.PerspectiveCamera(45, window.innerWidth /
window.innerHeight, 1, 2000); // configure camera settings & default
position
```

```
    camera.position.z = 20;
    camera.position.y = 2;

    // scene

    scene = new THREE.Scene();

    var ambientLight = new THREE.AmbientLight(0xcccccc, 0.4); // create 
light so simulation is actually visible
    scene.add(ambientLight);

    var pointLight = new THREE.PointLight(0xffffff, 0.8);
    camera.add(pointLight);
    scene.add(camera);

    var loader = new STLLoader();

    var material = new THREE.MeshPhongMaterial({ color: 0xffffff }); // 
blank white partially-reflective object texture by default

    loader.load('objects/assembly.stl', function (geometry) { // load .stl 
model

        assembly = new THREE.Mesh(geometry, material); // initialize object 
variable using stl model data
        scene.add(assembly); // insert object in visible scene

    });

    //

    renderer = new THREE.WebGLRenderer(); // configure WebGL renderer that 
turns scene objects into visual elements
    renderer.setPixelRatio(window.devicePixelRatio);
    renderer.setSize(window.innerWidth, window.innerHeight);
    container.appendChild(renderer.domElement);

    // orbit controls

    controls = new OrbitControls(camera, renderer.domElement); // sets up 
orbiting controls
    controls.target.set(0, 0, 0);
    controls.update();
    controls.enablePan = false;
    controls.enableDamping = true; // "inertia scrolling" but when dragging 
(continues moving in some cases even after dragging stops)

    window.addEventListener('resize', onWindowResize, false); // react to 
resized window

}
```

```
function onWindowResize() { // adjust camera & renderer settings

    camera.aspect = window.innerWidth / window.innerHeight;
    camera.updateProjectionMatrix();

    renderer.setSize(window.innerWidth, window.innerHeight);

}

//

function animate() {

    requestAnimationFrame(animate); // begin animation loop
    controls.update(); // check status of mouse & update camera accordingly
    render();

}

function render() {

    // TODO: Change camera target here

    controls.target.set(targetPos.x, targetPos.y, targetPos.z); // sets
camera target to coordinates indicated in targetPos object

    // TODO: change object positions, rotations, states, etc here - code
logic

    renderer.render(scene, camera); // after all code logic has run, render
scene for user viewing
}

function clip(input, limit1, limit2) { // restrict value of an input to a
certain range
    if (limit1 != null && input < limit1) {
        return limit1;
    } else if (limit2 != null && input > limit2) {
        return limit2;
    } else {
        return input;
    }
}

function submitInputs() { // called on start button click
    document.getElementById("output-text").style.visibility = "hidden"; //
hides output text

    input1 = clip(input1Input.value, input1Min, input1Max); // retrieves
inputs & restricts to within min/max values
    input2 = clip(input2Input.value, input2Min, input2Max);
    sendValues();
}

function sendValues() {
    input1Input.value = Math.round(input1 * 100) / 100; // if input value
was clipped, return updated value to inform user about what values the
simulation is actually using
    input2Input.value = Math.round(input2 * 100) / 100;
}

function sendOutput() {
    document.getElementById("output-text").style.visibility = "visible"; //
make output visible
    document.getElementById("output").innerText = output; // set output to
text from output string variable (would be set in render()    sendValues();
}

function sendValues() {
    input1Input.value = Math.round(input1 * 100) / 100; // if input value
was clipped, return updated value to inform user about what values the
simulation is actually using
    input2Input.value = Math.round(input2 * 100) / 100;
}

function sendOutput() {
    document.getElementById("output-text").style.visibility = "visible"; //
make output visible
    document.getElementById("output").innerText = output; // set output to
text from output string variable (would be set in render()
}
```

4 EXISTING SIMULATIONS

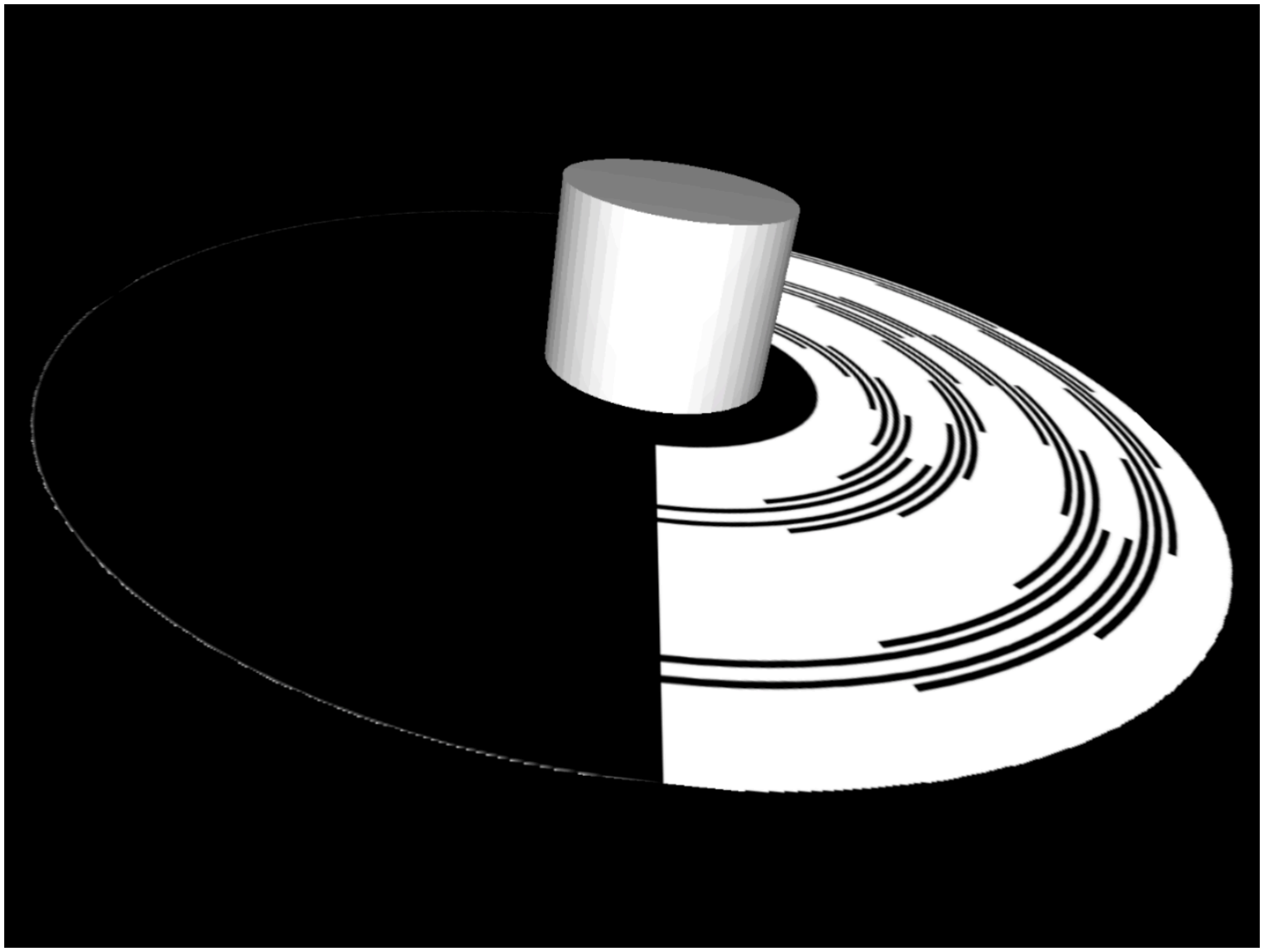
RAFT was presented with a total of 10 completed kit simulations at the end of the Eagle Project. These simulations were of the following kits:

- Black and White Makes Color
- Floating Compass
- Gravity Defying Frog
- Hovercraft
- Leonardo's Arched Bridge
- Roller Racer
- Scallop String Art
- Tongue Depressor Harmonica
- Waterbeads
- Zippy Catapult

This document includes an overview of each simulation, beginning on the next page.

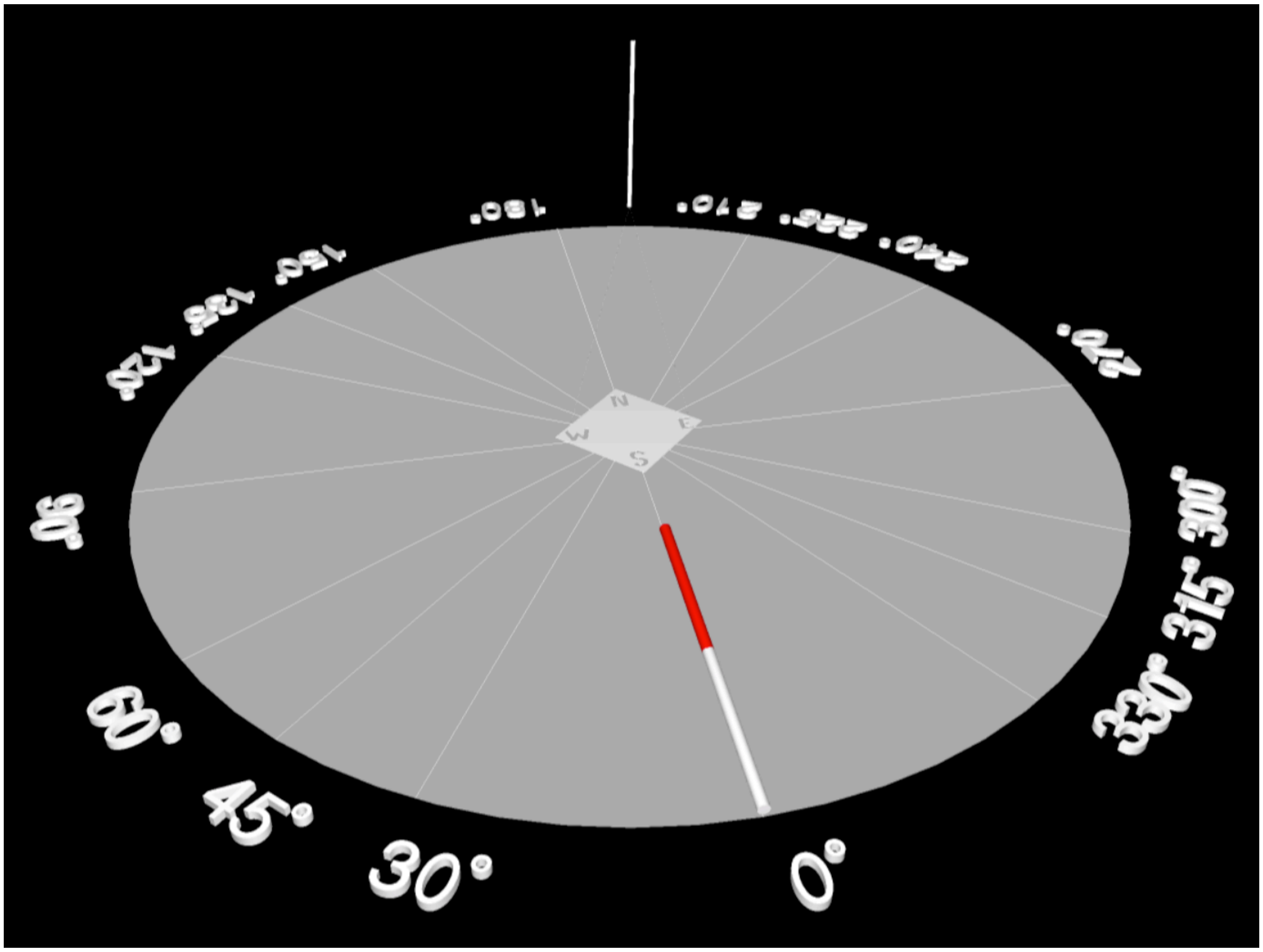
4.1 BLACK AND WHITE MAKES COLOR

The physical kit involves taping a circular piece of paper with this half-black, half-striped pattern to a CD and spinning it on a top to generate an optical illusion of colored circles. The simulation mimics this effect by replacing the black-and-white stripes with colored stripes when spinning at a speed above a certain threshold. As it slows down and drops below that threshold, the pattern reverts to black & white and the top begins to tip over and fall. There is no output value.



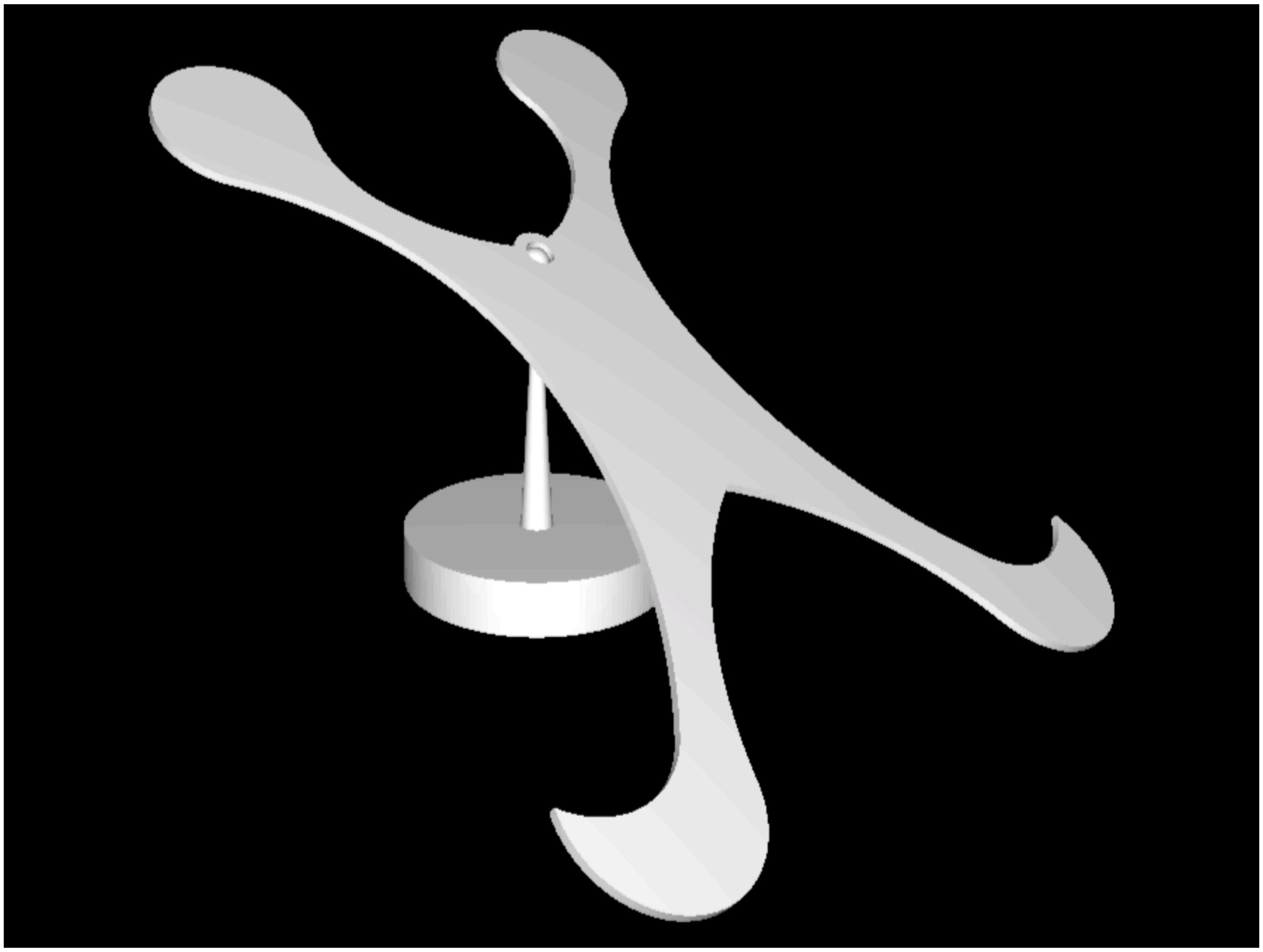
4.2 FLOATING COMPASS

The physical floating compass kit uses a magnet to orient a "compass" floating on water (taped to a metal needle), then allows it to rotate and align with the Earth's magnetic field. This simulation takes the magnet's angular position about the origin as an input, then moves the magnet to that rotated position and rotates the compass to follow. The user also has the option to rotate the magnet a full 180°, so that its south pole is facing the compass (by default, the north pole is). Below the magnet and compass, there is a disk labeled with common angles for reference. There is no output value.



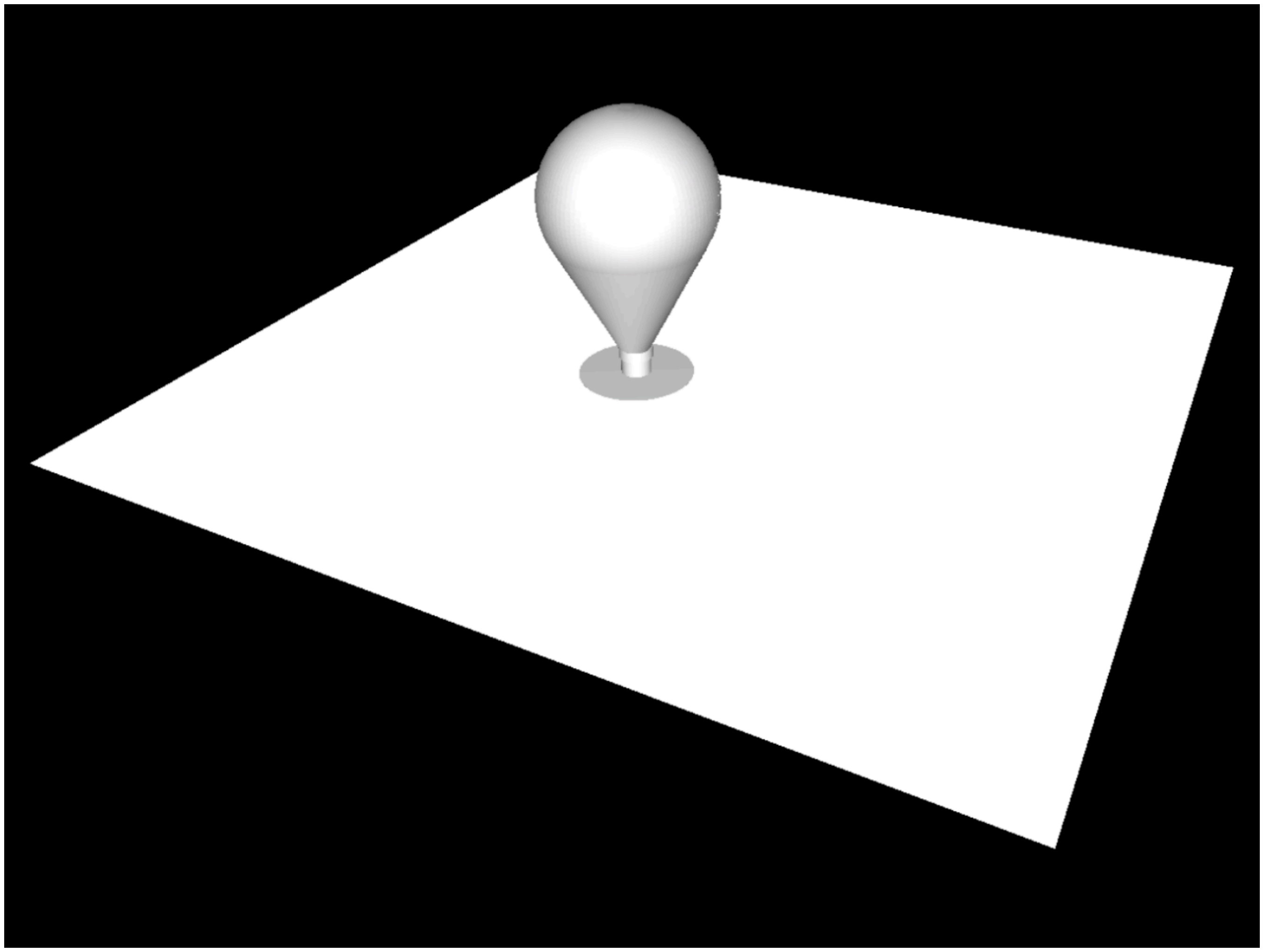
4.3 GRAVITY DEFYING FROG

The physical Frog uses paperclips on its “hands” and “feet” to balance and become level. This action is replicated by the user selecting a certain number of 0.5g paperclips to go on each limb, and the frog object rotating about the head of a pin to move to the respective orientation. There is no output value.



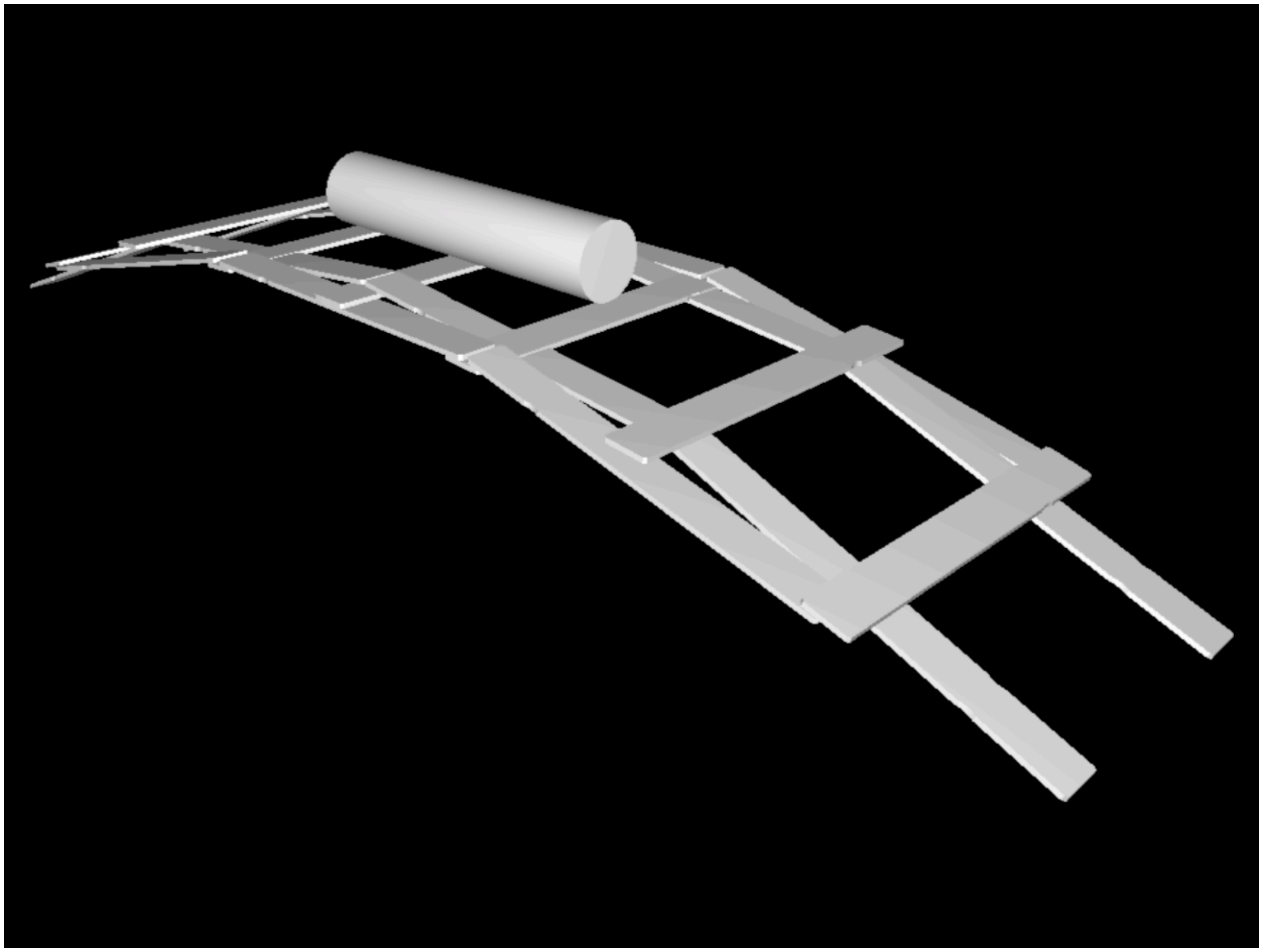
4.4 HOVERCRAFT

The physical Hovercraft kit includes a balloon to be attached to a disk base. When the balloon is filled, the base keeps it upright and the air escaping out of the bottom of the balloon causes the entire assembly to glide over a hard surface (like a tabletop). The simulation estimates this effect with an air pressure input in PSI and a “balloon” that hovers until it “runs out” of air (while simultaneously shrinking into nothing). The output value is the estimated hover time.



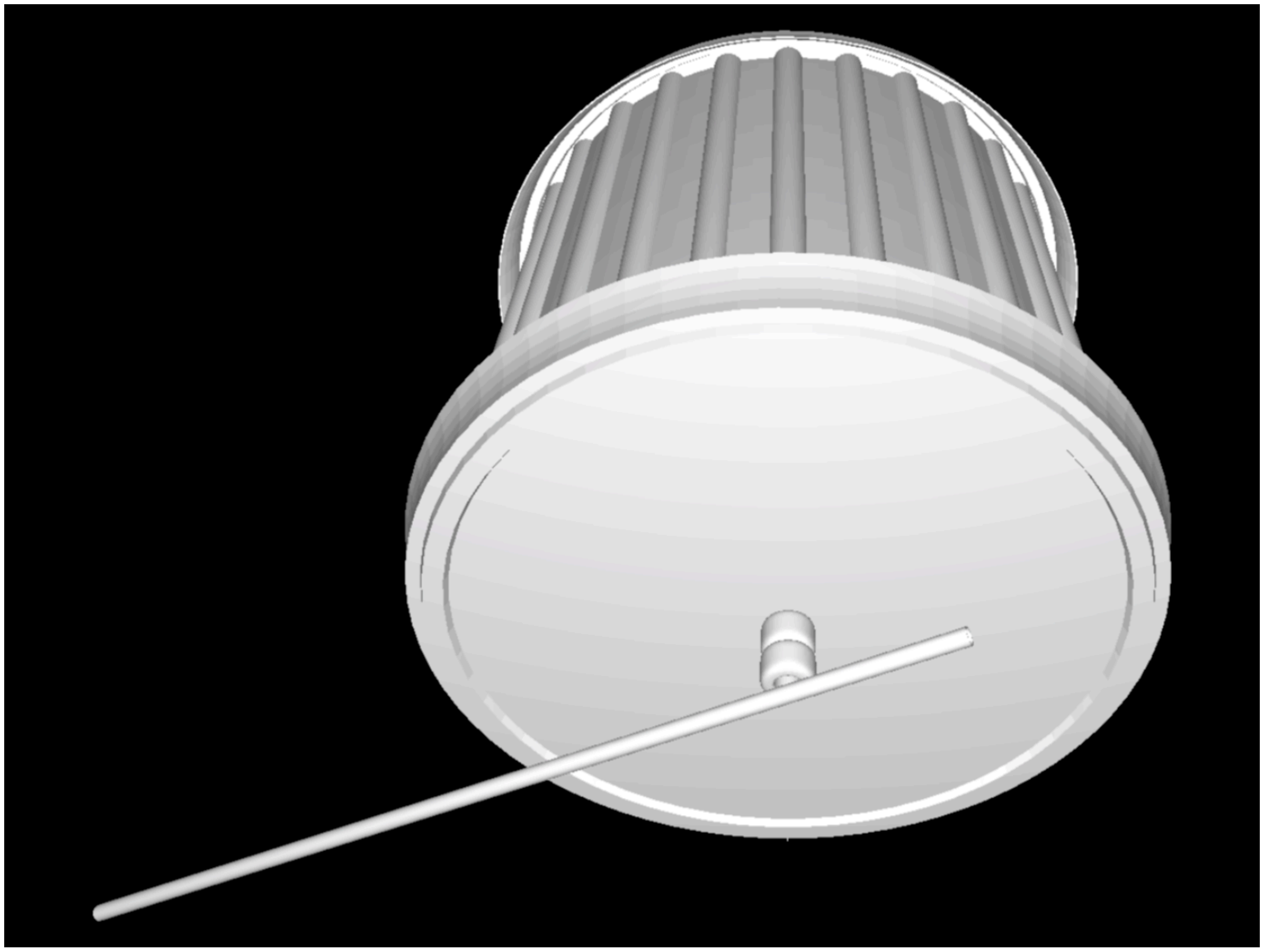
4.5 LEONARDO'S ARCHED BRIDGE

The physical kit is made from 18 wooden sticks (similar to popsicle sticks, but larger) interlocked in a bridge. This is replicated in the simulation, with cylindrical weights (0.5kg each) and rectangular sticks forming a 0.22kg bridge. After the user chooses a number of weights, they are generated and positioned in a grid on top of the structure, which becomes shorter and wider as if being compressed. There are two outputs: the change in the bridge's height from the initial position and the efficiency ratio (combined cylinder weight divided by total bridge weight).



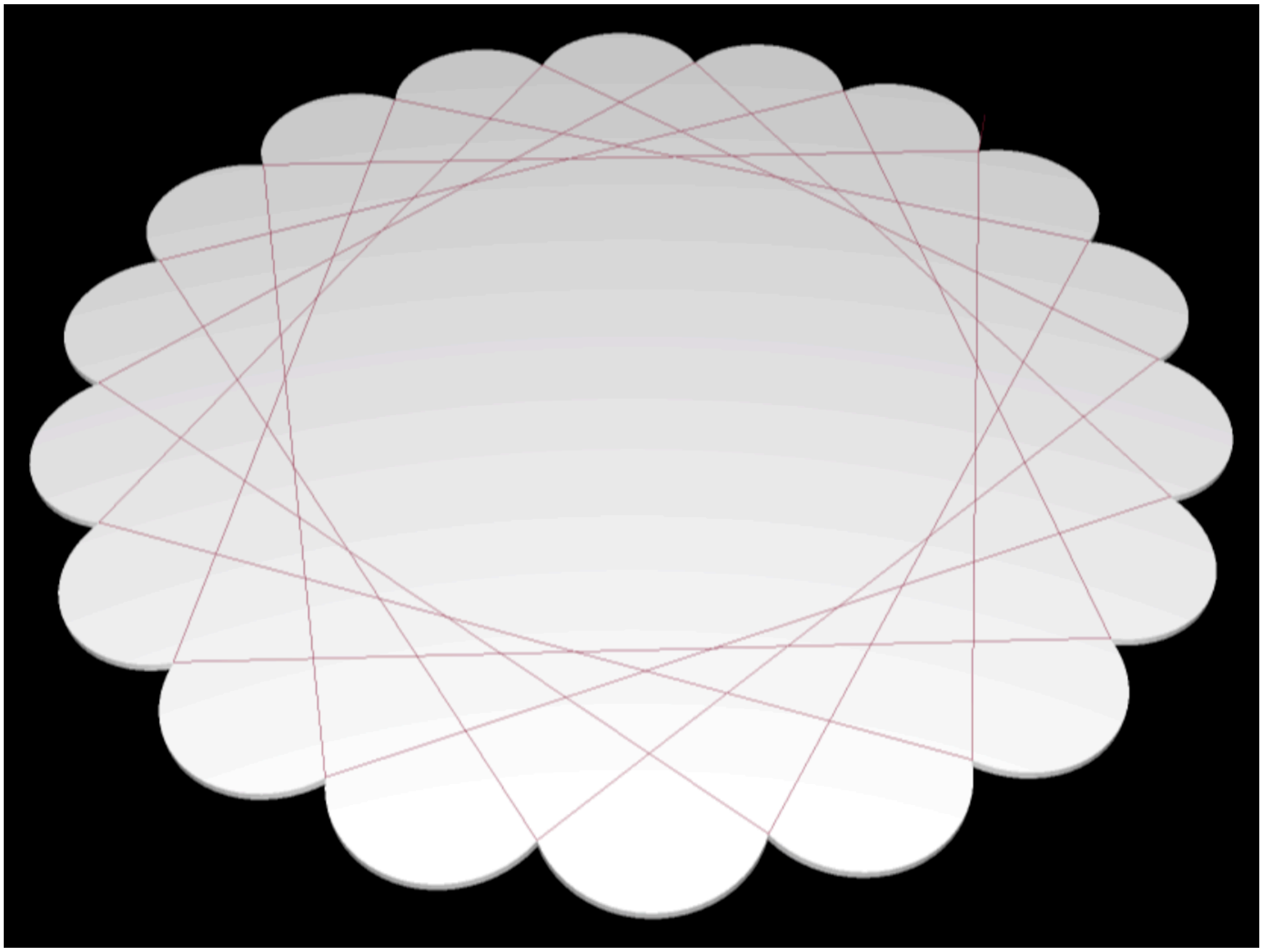
4.6 ROLLER RACER

The physical kit is propelled by rubber bands strung inside the hollow cylindrical shape, which store rotational energy from a person rotating the straw about the cylinder. When they release the straw, it spins until it hits a solid surface, at which point the cylinder's friction with the ground and the remaining stored energy in the twisted rubber band cause it to roll in the direction of the shorter end of the straw. Essentially, the kit becomes a rubber band-powered wheel. This functionality is replicated in the simulation: the user selects/enters a number of rotations (to two decimal places), the straw spins that amount clockwise then spins counterclockwise until it hits the "ground", and the kit rolls to the user's right (as observed from the default camera position). The output is the distance traveled from the starting line to the endpoint, and the camera target constantly updates to be halfway between the starting line and the kit's current position.



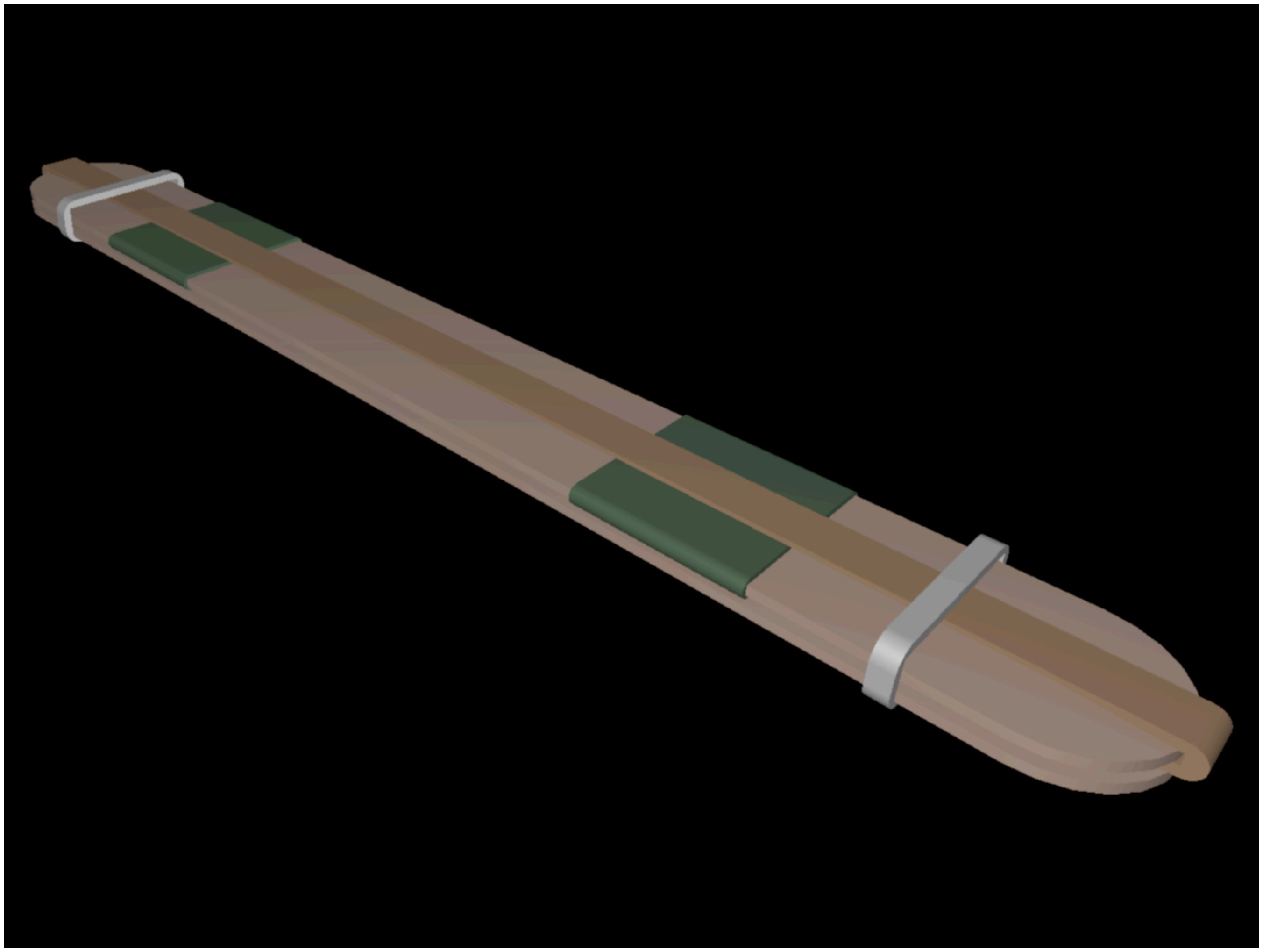
4.7 SCALLOP STRING ART

The physical kit essentially involves connecting the indents in the base with string to form different patterns on both the top and bottom. The simulation does this as well, generating a randomly-color for the string upon loading and taking user input to change the color as well as set the gap between each point of connection. On the top, the Scallop Gap selected is the number of semicircles between connections, and on the bottom, that number is increased by 1. There is no output.



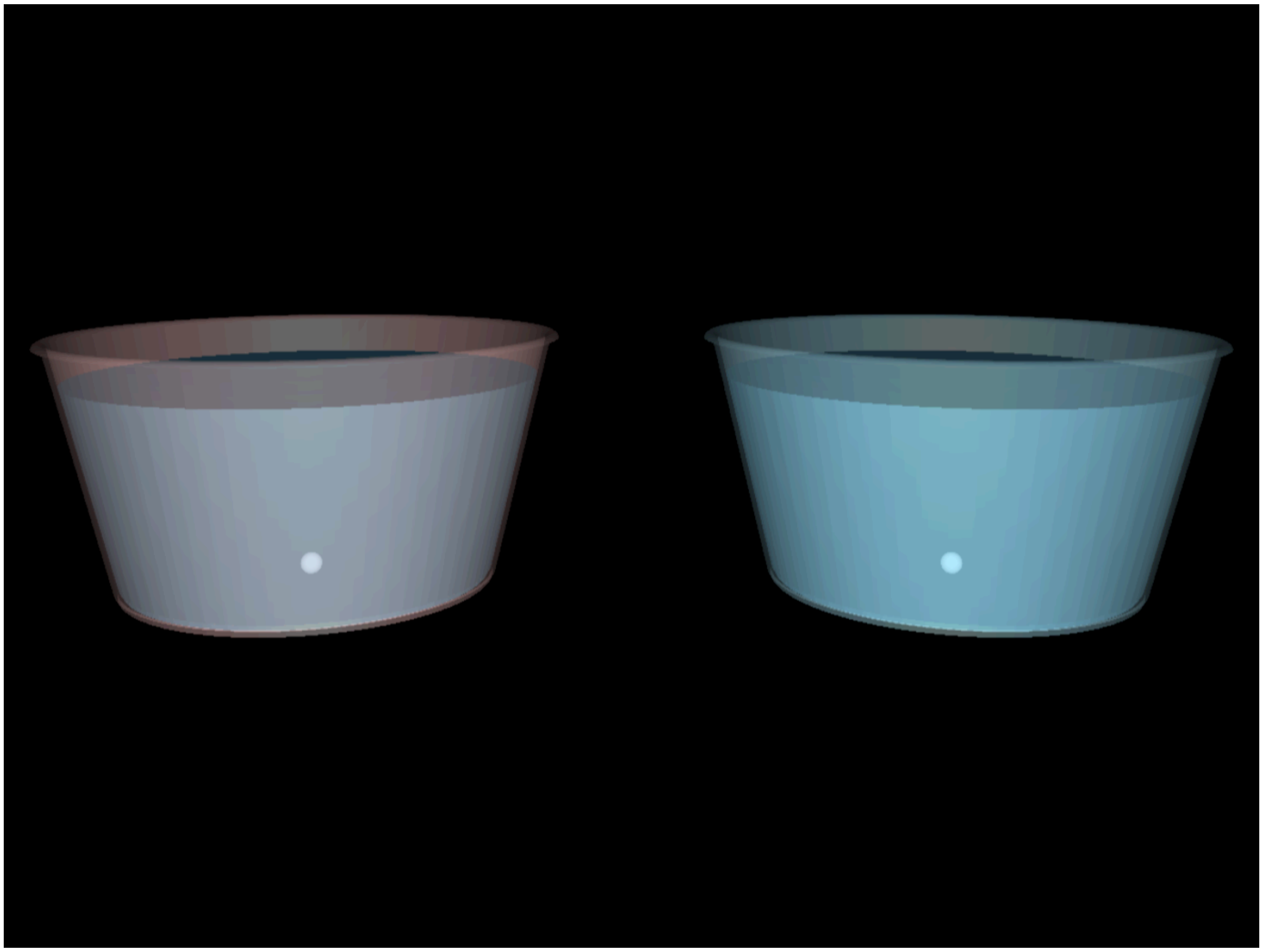
4.8 TONGUE DEPRESSOR HARMONICA

The physical kit varies the pitch of a harmonica by moving two paper sliders contacting a rubber band, which vibrates to create sound. The simulation gets user input on the gap between the sliders and uses three.js' built-in audio functionality to manipulate the audio file as well, however, this depends on a version of the HTML5 WebAudio API that is experimental and disabled by default in Safari and is not available in Internet Explorer or older versions of browsers like Google Chrome, Microsoft Edge, and Opera. If the user is in a browser that may not support the audio manipulation code, a popup warning will be displayed. There are two outputs in a table (on the opposite side of the screen from outputs in other simulations): the frequency of the sound and the exact note (in musical notation).



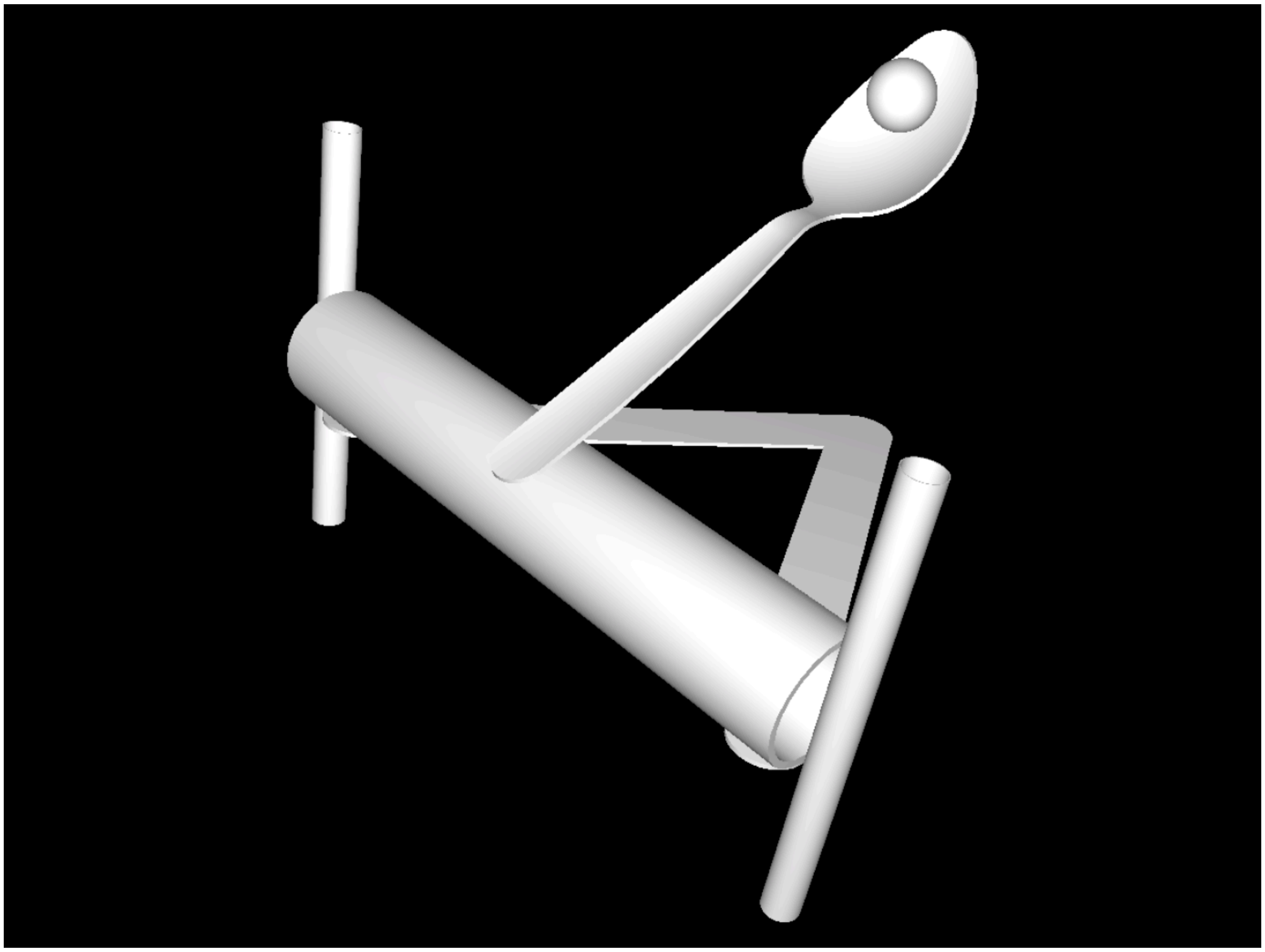
4.9 WATERBEADS

The physical kit involves differently-colored beads that grow when submerged in water and, if completely transparent, may "disappear" when underwater. The growth rate is affected by water temperature and salinity, so the simulation's inputs are two independent temperatures (for two beads in separate cups) in °C, amount of salt in grams (this amount would be applied to both cups), and the time to leave the beads to sit. When the simulation runs, the beads grow at their respective rates, then transfer to a test tube where the transparent one will disappear. The other bead, meanwhile, has a random color. There are two outputs: the diameters of each bead in millimeters after being submerged.



4.10 ZIPPY CATAPULT

The physical kit is a catapult that uses the flexibility of a plastic spoon as a catapult, raised using two plastic straws and reoriented by rotating the spoon about a cardboard tube. The simulation takes the height of the straws and the rotation of the spoon as inputs, as well as the initial velocity of the ball and gravity constant. When the user clicks “Start”, the spoon rotates and the straws slide up or down (as necessary), reaching the final position at the same time. The ball then launches at the given velocity, its position calculated using projectile motion equations. If the initial velocity was 0, the ball drops straight down, and if it is too low to pass the catapult, it will launch with its calculated trajectory and fall through the kit. Regardless of the velocity, the camera focuses on the center point between the ball and the front of the catapult, and the ball always “hits the ground”. If the ball does make it beyond the catapult, the simulation outputs the distance it traveled.



5 CREATING A NEW SIMULATION

Detailed instructions on creating a new simulation are available on RAFTsim's GitHub repository page or in the Maintenance Document included in this Eagle Project software package. At a high level, new simulations require CAD models of the kit exported to .obj or .stl files, a folder created by duplicating the appropriate template, inputs and output(s), and a general understanding of how the looped render() method will use the inputs to manipulate the kit's models and produce output(s).