Javascript: Badminton Man

Deborah Chang

December 9, 2016

Abstract

This project uses Javascript to create a 3D model of a badminton player. The badminton player then demonstrates the *clear*, a key badminton move.

1 Background

I have been playing badminton since childhood. Later, I formally trained for 3 years during my high school career. For my project, I create a Blobby Man, which represents a human figure and can be animated to perform various moves. I use the Blobby Man to demonstrate one of the basic moves in the game of badminton: the *clear*. This move allows a player to hit a birdie, the ball in a badminton game, to the far side of a badminton court and recover to the ready position to prepare for a return shot. This move is crucial in badminton, and mastering it can lead to great success in competition.

2 Overview

For this project, I create an animated figure through Javascript, similar to Ramya Babu's 2013 SwimmingMan [1]. This animated figure is known as a *Blobby Man*, which is a concept invented by Jim Blinn [2]. Most other Blobby Man projects, like the Running Guy [3] and Biking Man [4] were done in VPython. I chose to do this project in Javascript because I am concurrently learning it in another class (CS 105) and preferred to learn only one first programming language.

The player starts out standing (right before the birdie is to be served) and then moves into the clear. This includes both the arm movements and footwork. I also include a birdie to simulate an actual hit. For future expansion on this project, other moves, such as smashing or serving (either doubles or singles style), as well as a second player, could be included.

For this project, I create my own code while learning from a past project (i.e. Ramya's Swimming in 3D). I also go on the three.js library, which provides tools for creating and moving 3D objects in Javascript [5]. These include the variables I set for scene, camera, and renderer in order to actually create the objects I needed for the body of my badminton man. I also set up a hierarchy (or scene graph), for the body.

After I create my badminton man, I add in the code for animation. After this, I add code in that slowly rotates the scene (i.e. viewers can see the badminton man perform the clear from different directions).

3 The Clear Move

This move is essential to any player's arsenal of skills. It is often the first move that novices will be taught. The following description is applicable to right-handed players only; left-handed players perform these moves on the opposite side.

The first step is to rotate to the right. A player will first be facing forwards, and then he will step back with his right foot, so his feet are a bit more than a shoulder-length apart. His left foot will stay pointed forwards, while his right foot is now behind their left foot and facing outwards, away from their body, perpendicular to the left foot's direction. Both knees will be bent at this point, with weight shifted to the right foot. Concurrently, the player will raise his left arm up in front of him at a diagonal. His arm will be roughly 45 degrees above the horizontal. He will also raise his right arm up, bending his elbow behind his head to form a "V" shape, where his right elbow will be near his right ear[1].

As the birdie (shuttlecock) approaches, the player will begin to straighten his right arm as vertically as possible by his ear in a circular fashion, his left arm will begin to drop down towards the body, and he will prepare to switch his feet. The player should hit the birdie at the highest point possible, where his right arm is as vertical as possible. As the racket makes contact with the birdie, the player will snap his wrist downwards, and his feet will be midair as the right foot goes forward and the left foot goes backwards. After hitting the birdie, the player's right arm will continue to move in a downwards arc, approaching his body. He will land softly with his right foot in front and left in back, and bend his knees to absorb the impact.

4 Hierarchy

In the Blobby Man scene graph, there is a hierarchy of frames. As can be seen in Figure 1, moving in a top-down fashion, each successive body part is linked to the previous body part. So the left foot is linked to the left leg, which is linked to the left knee, etc[2] . This is important to keep in mind when composing the code for animation in Javascript. In order to create this hierarchy, where each object (body part) is linked to another to create the overall, or parent, object (the entire Blobby Man), I have to add (link) them together. The code in Figure 2 shows what it means to add these objects in Javascript.

The point of doing this is so that if I animate the left pelvis, for example, the rest of the leg (the left thigh, left knee, and left leg) moves in conjunction with the left pelvis.

[1]Beginners often start directly from here and only perform the next few steps.

[2]Left refers to the viewer's left, **not** the Blobby Man's left.

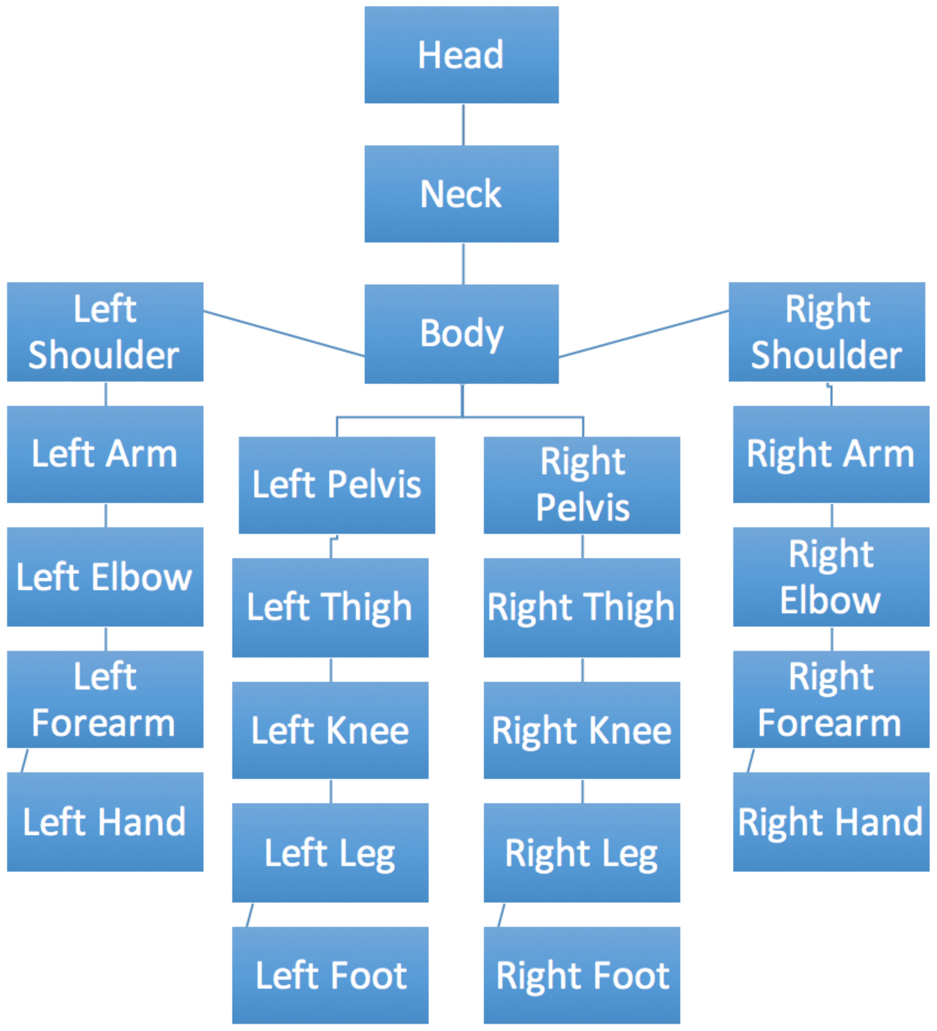
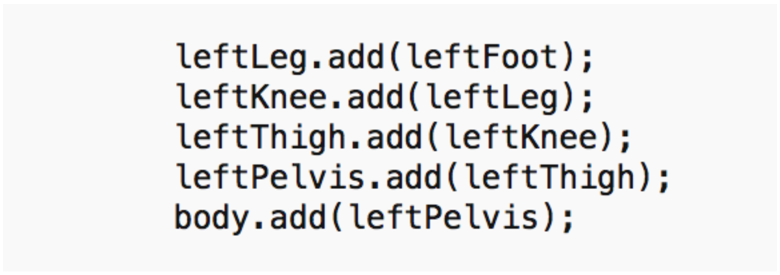


Figure 1: Hierarchy for a Blobby Man

This way there is not a random body part left hanging in mid-air while the rest of the body moves to a different location. This is related to how the human body actually moves. When a body part that is higher up in the hierarchy is moved, then every body part below the parent body part will also move. However, if I were to move my left foot, then only my left foot would move. This is because the left foot is at the lowest position of the hierarchy. The same concept would apply for, say, my right hand.



```
leftLeg.add(leftFoot);
leftKnee.add(leftLeg);
leftThigh.add(leftKnee);
leftPelvis.add(leftThigh);
body.add(leftPelvis);
```

Figure 2: Code for creating hierarchy

5 Animation

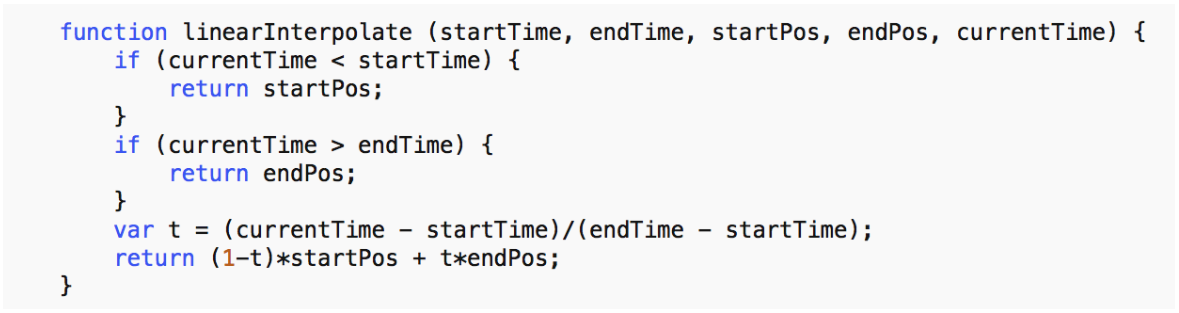
5.1 Linear Interpolation

In order to animate the Blobby Man, I started with the following linear interpolation function:

$$f(t_0, t_1, p_0, p_1, t) = (1 - \frac{t - t_0}{t_1 - t_0})p_0 + \frac{t - t_0}{t_1 - t_0}p_1$$

where t_0, t_1, p_0, p_1, and t correspond to start time, end time, start position, end position, and current time, respectively.

This can also be seen in Figure 3.



```
function linearInterpolate (startTime, endTime, startPos, endPos, currentTime) {
    if (currentTime < startTime) {
        return startPos;
    }
    if (currentTime > endTime) {
        return endPos;
    }
    var t = (currentTime - startTime)/(endTime - startTime);
    return (1-t)*startPos + t*endPos;
}
```

Figure 3: Linear Interpolation

The first line of code to be executed by the `linearInterpolate` function occurs if the `currentTime` parameter is before the `startTime`. Then the function `linearInterpolate` returns the `startPos` (start position); if the `currentTime` parameter is after the `endTime`, then the the function returns the `endPos` (end position). If the `currentTime` is not within the parameters of `startTime` and `endTime`, then running the last two lines of code would not return the desired Blobby Man movement. Therefore, if either of these two return statements are implemented, the rest of the function is not implemented.

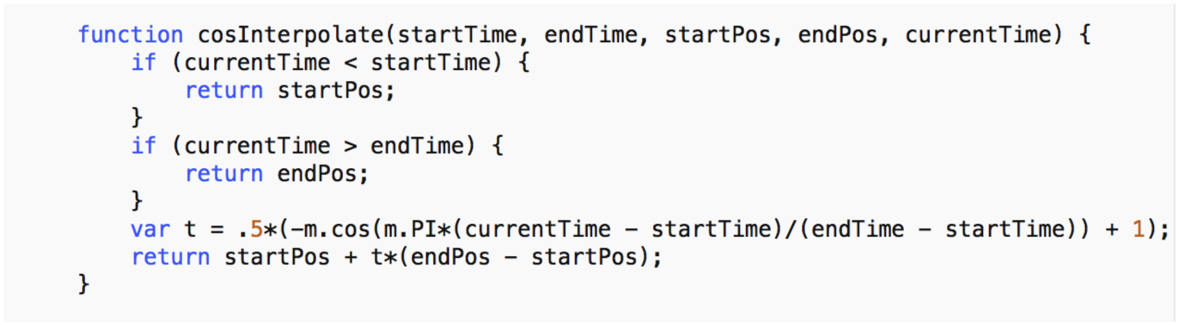
If the `currentTime` is within the `startTime` and `endTime`, then the last two lines of code are executed by the function `linearInterpolate`. The variable `t` will give a value between 0 and 1. This is a fraction, or proportion, of how far along the Blobby Man is in its movement from `startPos` to `endPos`. If the Blobby Man has not begun to move, then `t` will be 0 since `currentTime-startTime` would be 0. With `t` as 0, multiplying `(1-t)`, or 0, to `startPos` and then adding this to (`t *endPos`), or 0, will return the value of `startPos`. Then as `currentTime` approaches `endTime`, and the Blobby Man is in motion, `t` will be equal to values greater than 0 and approaching 1. Multiplying `(1-t)` to `startPos` and then adding `t*endPos` will return the position the Blobby Man is in relative to the `startPos` and `endPos`. When `t` is 1, `(1-t)*startPos` is 0 and `t*endPos` equals `endPos`. This means the function will return `endPos` (since adding 0 to `endPos` equals `endPos`), and the Blobby Man will have completed its movement from `startPos` to `endPos`.

5.2 Sinusoidal Interpolation

Using the `linearInterpolate` function to animate the Blobby Man creates the correct movements, but it is not conducive to smooth movements. Therefore, I create the following function for sinusoidal interpolation:

$$f(t_0, t_1, p_0, p_1, t) = p_0 + (0.5(-\cos(\pi(\frac{t - t_0}{t_1 - t_0})) + 1))(p_1 - p_0)$$

This can also be seen in Figure 4. This function is similar to linear interpolation,



```
function cosInterpolate(startTime, endTime, startPos, endPos, currentTime) {
    if (currentTime < startTime) {
        return startPos;
    }
    if (currentTime > endTime) {
        return endPos;
    }
    var t = .5*(-m.cos(m.PI*(currentTime - startTime)/(endTime - startTime)) + 1);
    return startPos + t*(endPos - startPos);
}
```

Figure 4: Sinusoidal Interpolation

except that there is a cosine function, rather than a linear function. The fraction `(currentTime - startTime)/(endTime - startTime)` still returns a value between 0 and 1; in this case, this value is also multiplied by π, since I am using a cosine function. Since `(currentTime - startTime)/(endTime - startTime)` will only return values between 0 and 1, the domain of the cosine function only runs from 0 to π. Therefore, the cosine function will only return values in the range from -1 to 1 as seen in Figure 5.

I then multiply this cosine function by -1 so that the cosine curve is mirrored about the x-axis, and I add 1 so that the curve is translated up by 1, and will therefore have a range of values between 0 and 2 (and will not return negative values). This can be seen in Figure 6.

I then multiply this by 0.5 so that the variable `t` will only return values between 0 and 1, rather than 0 and 2, since I want `t` to be a fraction of the distance between `startPos` and `endPos`. This can be seen in Figure 7.

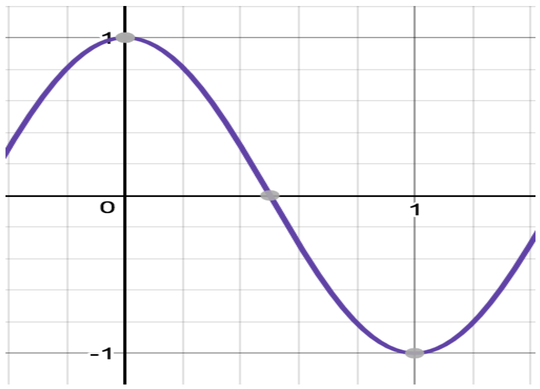


Figure 5: Cosine graph showing range from -1 to 1

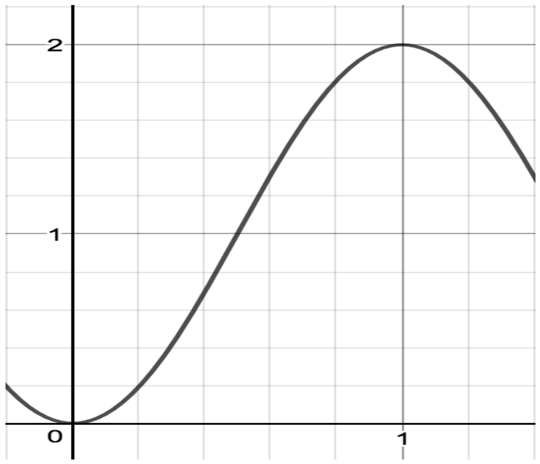


Figure 6: Cosine graph showing range from 0 to 2

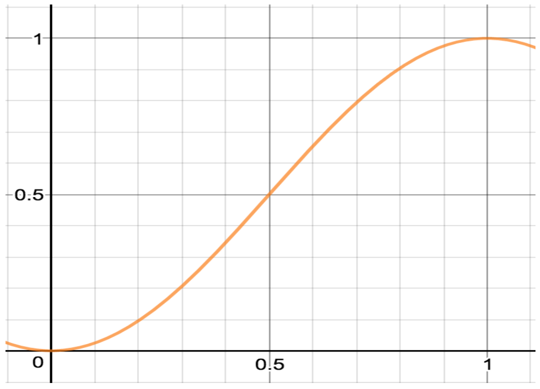
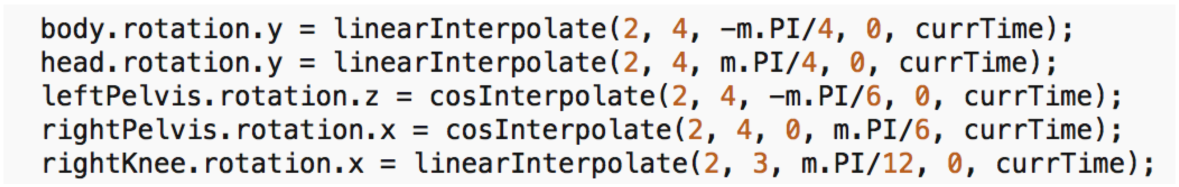


Figure 7: Cosine graph showing range from 0 to 1

In the case of sinusoidal interpolation, the movement of the Blobby Man occurring between `startPos` and `endPos` will mimic a cosine curve, rather than the linear curve of the linearInterpolate function. The Blobby Man movements will be smoother, since the movement follows along the curve of a cosine function rather than the straight line of a linear function.

I use both the `linearInterpolate` function and the `cosInterpolate` function to animate my Blobby Man. A snippet of this code, with parameters passed in for `startTime`, `endTime`, `startPos`, `endPos`, and `currentTime`[3], can be seen in Figure 8.

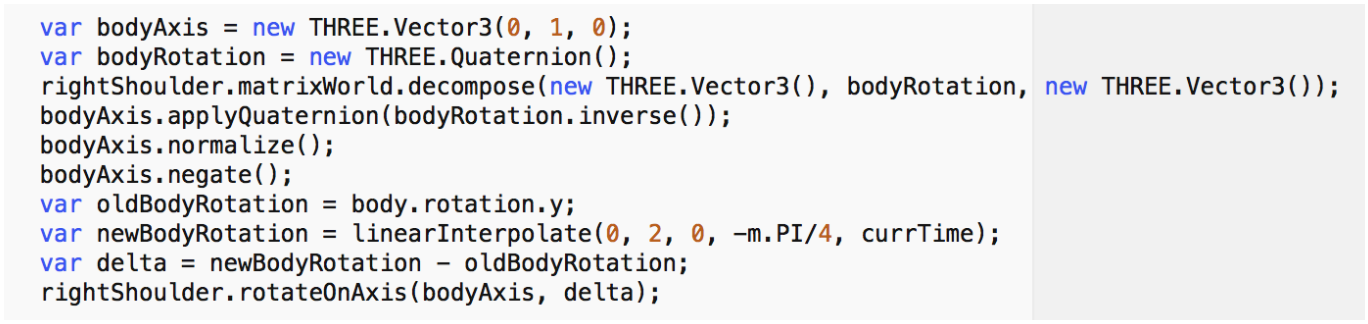


```
body.rotation.y = linearInterpolate(2, 4, -m.PI/4, 0, currTime);
head.rotation.y = linearInterpolate(2, 4, m.PI/4, 0, currTime);
leftPelvis.rotation.z = cosInterpolate(2, 4, -m.PI/6, 0, currTime);
rightPelvis.rotation.x = cosInterpolate(2, 4, 0, m.PI/6, currTime);
rightKnee.rotation.x = linearInterpolate(2, 3, m.PI/12, 0, currTime);
```

Figure 8: Code for linear and sinusoidal interpolation

5.3 Preventing Right Shoulder Rotation

I rotate the Blobby Man's body to the right, as occurs on the badminton court. This leads to a new issue where I do not want the `rightShoulder`[4] to rotate with the body. In order to do this, I implement the code seen in Figure 9.



```
var bodyAxis = new THREE.Vector3(0, 1, 0);
var bodyRotation = new THREE.Quaternion();
rightShoulder.matrixWorld.decompose(new THREE.Vector3(), bodyRotation, new THREE.Vector3());
bodyAxis.applyQuaternion(bodyRotation.inverse());
bodyAxis.normalize();
bodyAxis.negate();
var oldBodyRotation = body.rotation.y;
var newBodyRotation = linearInterpolate(0, 2, 0, -m.PI/4, currTime);
var delta = newBodyRotation - oldBodyRotation;
rightShoulder.rotateOnAxis(bodyAxis, delta);
```

Figure 9: Code for preventing `rightShoulder` rotation

I create the vector `bodyAxis` with x, y, and z coordinates of 0, 1, and 0. Then I create a variable `bodyRotation` to hold information about an object's transformation, namely its translation, rotation, and scaling components. This information is retrieved through the `Quaternion` method. Then I retrieve the `rightShoulder`'s transformation information from world space, meaning the x, y, and z components of the `rightShoulder`'s transformation orientation are based on the world space. I then `decompose`, or separate, the transformation into its three separate types: translation, rotation, and scaling. The rotation is represented by `bodyRotation`, while translation and scaling are represented by vectors as I do not need the information for these two types of transformation. Then I apply the inverse translation, or a translation

[3]The argument `currentTime` is simply replaced by a variable called `currTime` that retrieves the time that has elapsed since January 1, 1970 00:00, which is known as the epoch time.

[4]Again, right refers to the viewer's right, which is the Blobby Man's left

in the opposite direction, of the `rightShoulder` to the vector `bodyAxis`. I normalize `bodyAxis` to make sure the vector is a unit vector and then negate it. Negating the vector inverts, or flips, it, meaning when I rotate the `rightShoulder` around this vector in the last line, the `rightShoulder` will rotate in the correct direction. For example, instead of rotating to the left as an object would without negating the vector, an object would rotate to the right, since the vector has been inverted. I do this so that the `rightShoulder` will rotate in the opposite direction as the body, and, therefore, it will visually appear as if the right shoulder has remained in place. I create the variables `oldBodyRotation` and `newBodyRotation`, and then I create the variable `delta` as the difference between the new and old rotations. This difference is the amount I then rotate the `rightShoulder` by around the `bodyAxis` vector.

This creates the appearance that the rest of the body is rotating to the right, while the `rightShoulder` is not.

6 References

1. Babu, Ramya. "Visualizing the Dynamics of Swimming in 3D with HTML and Javascript." *Ramya Babu, Math198 Home Pages.* N.p., 16 Dec. 2013. Web. 16 Nov. 2016.

2. Jim Blinn. *Jim Blinn's Corner: A Trip Down the Graphics Pipeline.* Morgan Kaufmann, 1996.

3. Piper, Matthew. "RunningGuy: A Framework Experiment in VPython." Matthew Piper, Math198 Home Pages. N.p., 15 Nov. 2011. Web. 16 Nov. 2016.

4. Smith, Gillian. "Bicycle Man: A Blobby Man Adventure." Gillian Smith, Math198 Home Pages. N.p., 4 Dec. 2012. Web. 16 Nov. 2016.

5. Cabello, Ricardo, and three.js Authors. *three.js.* three.js, threejs.org. Accessed 28 Nov. 2016.