

A Rails Web App to Replace the Traditional Whiteboard for Basketball Coaches

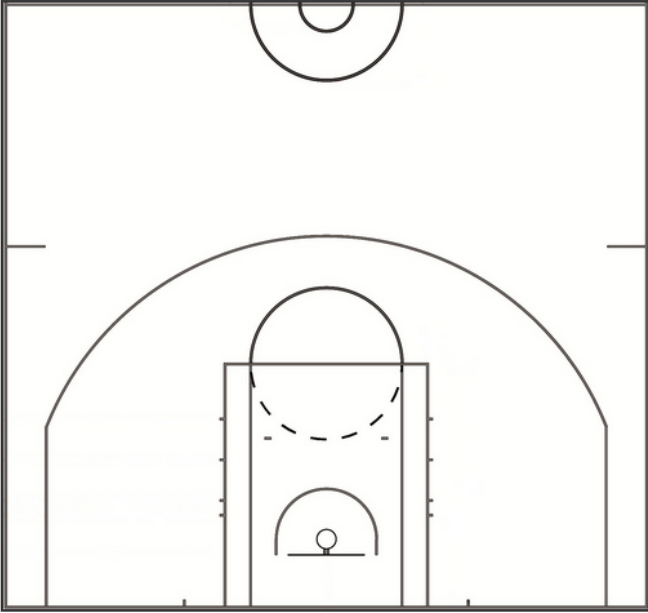
A Senior Project in Computer Science by Dimitri Cavoulacos
Advised by Holly Rushmeier

The original idea for this project was extremely vague. I knew I needed to come up with a project idea that was personally appealing, so the very first thing that came to mind was basketball. Then came the harder part of turning a theme into a project. I considered advanced statistical analysis, player performance projections and the like; but while I respect the stats for the insight they provide, only trusting the numbers creates a disconnect from the game of basketball. The idea for a whiteboard app for basketball coaches came from my personal relationship with the sport. I have played basketball longer than not, and what appeals to me the most about the game is the unique balance between individual and team performance. Great players like Michael Jordan, LeBron James and Kevin Durant can have a huge impact on the floor, but the best basketball is played when a team plays together in synchrony¹, like the 1986 Boston Celtics or the 1996 Bulls. I realized that great basketball comes as much from great coaching as great performance, and the idea for this project started to crystalize.

I chose to use Ruby on Rails for this project for three simple reasons: I was trying to build an app, I had taken a summer crash course on building web apps with Rails, and I didn't know any other way to build an app. I quickly realized that Rails didn't have a graphical display library, and initially planned on using a 2D game development library called Gosu to handle the visual aspect of the project. This became very complicated very quickly, so I decided to try to piece together what display functions I needed on my own. But first, I created the foundation for the application. The app needed to allow the user to create, update and design set basketball plays, or set plays, in which five players carried out a

¹ San Antonio Spurs Coach Gregg "Pop" Popovich is living proof of that.

sequence of actions that ultimately led to one player shooting the ball. Each set play would be composed of action frames, in which each player may carry out up to one action. These action frames would act as a timeline of sorts, separating sequential actions from concurrent ones. I created action frames, players and actions first. Each player belonged to an action frame, and each action belonged to both an action frame and to a player. I realized that by always creating five actions when an action frame is created, this sets the correct upper bound for simultaneous actions: a player can move once or not at all, but never twice in one action frame.



-1

Edit Player Type End x End y Teammate

[Edit](#) PG

[Edit](#) SG

[Edit](#) SF

[Edit](#) PF

[Edit](#) C

```
[#<Action id: 26, action_frame_id: 10, player_id: 6, action_type: nil, end_position_x: nil, end_position_y: nil, created_at: "2014-04-30 17:42:49", updated_at: "2014-04-30 17:42:49", teammate: nil>, #<Action id: 27, action_frame_id: 10, player_id: 7, action_type: nil, end_position_x: nil, end_position_y: nil, created_at: "2014-04-30 17:42:49", updated_at: "2014-04-30 17:42:49", teammate: nil>, #<Action id: 28, action_frame_id: 10, player_id: 8, action_type: nil, end_position_x: nil, end_position_y: nil, created_at: "2014-04-30 17:42:49", updated_at: "2014-04-30 17:42:49", teammate: nil>, #<Action id: 29, action_frame_id: 10, player_id: 9, action_type: nil, end_position_x: nil, end_position_y: nil, created_at: "2014-04-30 17:42:49", updated_at: "2014-04-30 17:42:49", teammate: nil>, #
```

Then I created set plays, and with set plays came a few issues. First of all, each action frame needed to have a sense of its place in set play's timeline. This meant including to associations, :next and :prev, which would link action frames to each other. A :prev value of -1 meant that the action frame was first in the set play, and a :next value of -1 meant that a shot occurred. The second issue was that players at this point still belonged to a single action frame, so the data structure of the application had to be changed. Players were made to belong to set plays instead to allow a given player to progress smoothly

through a set play instead of constantly updating the set play's current player variable. But this presented a third problem. Players initially belonged to action frames because it allowed the app to keep track of all the necessary information, i.e. where player A is standing at the end of an action frame, where player B is standing at the end of an action frame, who has the ball, etc. The idea was to have the players' position values and ball Boolean represent the state of the court at the beginning of the action frame, while the players' actions represent the state of the court at its end. By keeping the same five players through a set play, we lose the initial configuration of the team as the set play started, so to solve that problem, I created offenses.

Originally, I had planned to add two position variables (x and y) and one ball Boolean per player to the set play model, for a total of fifteen additional variables. The values for all these variables would be preset to represent existing basketball formations², and would have to be stored in the database, so I quickly realized that it would be much simpler to create a new model, offense, which would hold all of the values and would be linked to a set play upon creation. As an added benefit, this would allow set plays to be categorized by offense type which could allow for added functionality down the road, such as multiple "options", or action sequences, for a single set play. This solved the problem of the lost initial data, and I moved on from there to set play design.

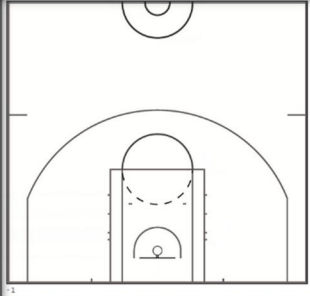
The first step was to enable the creation of an additional action frame. This required that five actions be created and associated with the appropriate players, that the new action frame be set to the old action frame's :next value and that in return the old be set to the new's :prev. The action frame on display then needed to be updated, so this required that I create a variable for set plays to store the current action frame. Furthermore, since the position of each player at the end of an action frame is stored in that action frame's actions, any blank actions would be set to "Move" types, with the final position being the same as the initial position. Finally, the position and ball values for all the players needed to be updated according to their action types, so a "Move" action type would update position and a "Pass" action type would

² The formations were found at http://hooptactics.com/Basketball_Basics_Offenses

update the ball Booleans of both the passer and the receiver. Once this was completed, moving forward one action frame became quite simple (it requires the same process, only without filling blank actions), so the next challenge was moving to a previous action frame.

As previously stated, previous player position and ball values are stored first in the offense and then in the following actions. However, this became quite complicated. When there are only two action frames, moving backwards is simple as the players reassume their initial configurations. When there are more than two action frames, it is still quite simple for any "Move" or "Screen" action types, as these only affect position values. So when there are three action frames and we wish to go from the third to the second, the players' positions are set by the actions of the first action frame. However, "Pass" action types were more problematic. Suppose now that there are four action frames and we wish to go from the fourth to the third. Suppose the player who started with the ball moves in the first action frame, passes in the second then moves again in the third to his current position. That player's position at the start of the third action frame will be stored not in the second action frame actions as that action was a pass, but in the first action frame actions, so we must go back three action frames to retrieve the position values. Every step of the process also requires a test for a :prev value of -1, in which case the relevant values would again be reset to the initial offense values. Once moving forwards and backwards through a set play's action frames was implemented, I quickly added a function to end a set play if a shot occurred. This function sets the action frame's :next value to -1 and sets a Boolean flag to true for extra measure. It then updates the current action frame value to the first action frame in the set play and sends the user back to the index page. With this functionality complete, I had the three biggest challenges left to overcome: streamlining the action mechanism, displaying the players and actions on the canvas and animating the action frames.

The action mechanism to this point was the Rails default setting: a link to an update form. Once the form was submitted, the app would redirect the user back to the set play page with the new data in place. However, the frequent redirection and page changes made for an unpleasant and disjointed experience in addition to inconveniencing the user by omitting any visual guidance while choosing a course of action for the set play.



Edit Player Type End x End y Teammate
☐ PG
☐ SG
☐ SF
☐ PF
☐ C

```

1
[#<action id: 26, action_frame_id: 10, player_id: 6, action_type: nil, end_position_x: nil, end_position_y: nil, created_at: "2014-04-30 17:42:49", updated_at: "2014-04-30 17:42:49", teammate: nil> #<action id: 27, action_frame_id: 10, player_id: 7, action_type: nil, end_position_x: nil, end_position_y: nil, created_at: "2014-04-30 17:42:49", updated_at: "2014-04-30 17:42:49", teammate: nil> #<action id: 28, action_frame_id: 10, player_id: 8, action_type: nil, end_position_x: nil, end_position_y: nil, created_at: "2014-04-30 17:42:49", updated_at: "2014-04-30 17:42:49", teammate: nil> #<action id: 29, action_frame_id: 10, player_id: 9, action_type: nil, end_position_x: nil, end_position_y: nil, created_at: "2014-04-30 17:42:49", updated_at: "2014-04-30 17:42:49", teammate: nil> #<action id: 30, action_frame_id: 10, player_id: 10, action_type: nil, end_position_x: nil, end_position_y: nil, created_at: "2014-04-30 17:42:49", updated_at: "2014-04-30 17:42:49", teammate: nil>]

```

Updating Action

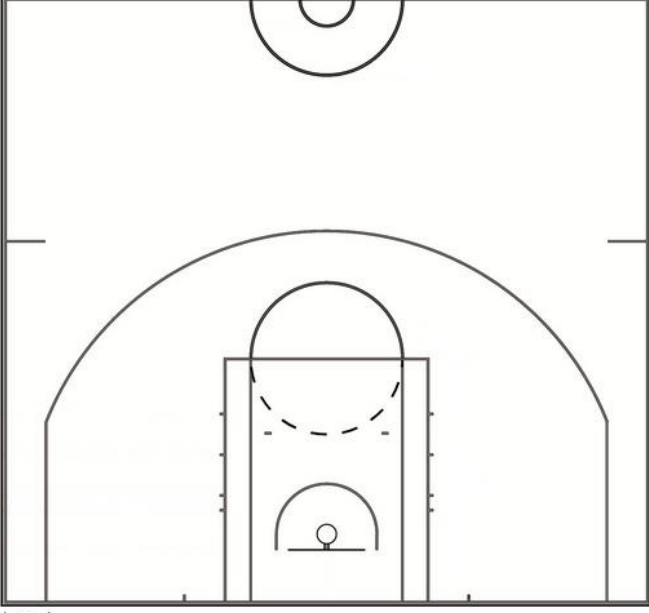
Editing action

End position x:
End position y:
Teammate:

Action Form

To remedy this, I created an action update form for each action in the set play page itself. The form took into account whether a player was in possession of the ball or not and displayed possible action types accordingly.

Name: Test 1132
Type: Single Post Four Out
[<#<ActionFrame id: 12, set_play_id: 7, prev_id: -1, next_id: nil, name: nil, ends_set: nil, created_at: "2014-04-30 19:39:04", updated_at: "2014-04-30 19:39:04">]
Current AF = 12



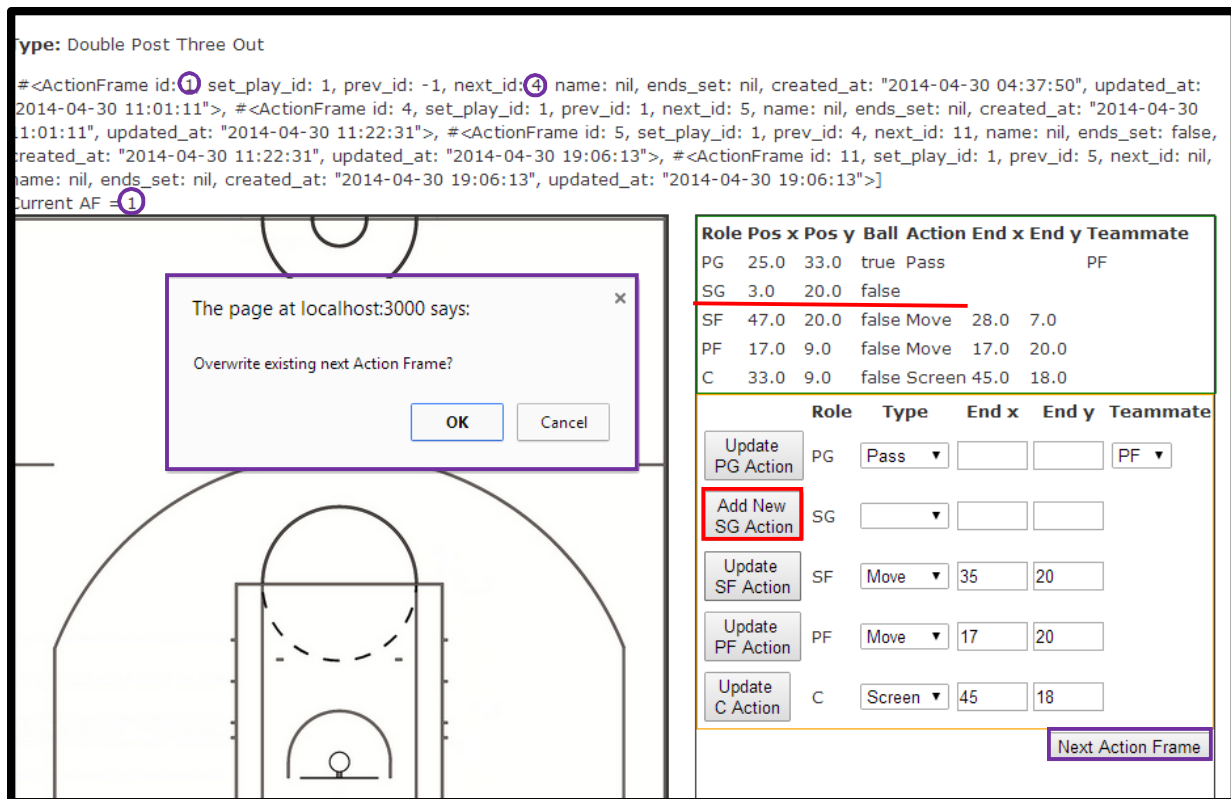
Role	Pos x	Pos y	Ball	Action	End x	End y	Teammate
PG	33.0	33.0	true				
SG	17.0	33.0	false				
SF	47.0	20.0	false				
PF	9.0	22.0	false				
C	33.0	9.0	false				

Add New PG Action
Add New SG Action
Add New SF Action
Add New PF Action
Add New C Action

Move
Screen
Pass
Shoot

Back

The buttons also take into account whether or not a non-trivial action already exists for a player, and display Add New or Update in the appropriate cases. Moreover, if a next action frame already exists, updating an action brings a pop-up confirmation, warning the user that their data will be overwritten.



The second of the three big challenges was to display the players and actions in the canvas, and I started with the players. I had been using the bottom-left corner of the court as my origin in setting all the position values, but HTML5 Canvas uses the top-left corner, so ran a translation to reconcile the two. From there, I turned to Javascript and created local position variables that could interact with the Canvas. All the x position values were multiplied by ten and all the y position values were multiplied by minus ten (since the origin was translated, not reflected), and the players' roles were displayed at the corresponding location. I then looped through the players to find the player with a true ball Boolean, and set x and y coordinates for the appropriate position. However, the players' roles were displayed by the fillText() method, which begins at a position (x, y) and fills leftward, 88 pixels per player role in this case. However, the circle used to represent the ball needs an origin point (x', y'). I found that x' was equal to x + 12 and y' was equal to y - 7. This became increasingly relevant when I reached animation.

pass does not move in the action frame, then the dashed arrow points straight to the player's position. However, if the recipient of the pass does move, then the arrow needs to point at where the player will be when they catch the pass. This is all established in the giant player loop if-else statement, and then all the information required for each action line is sent to the *draw_action()* method. This implements the arrowhead and dashed line drawing previously mentioned as well as the unique "Screen" action arrow and a small statement to avoid overlap between the player text and the arrows. This is especially true of passes when the recipient does not move, so the end positions must be tweaked.

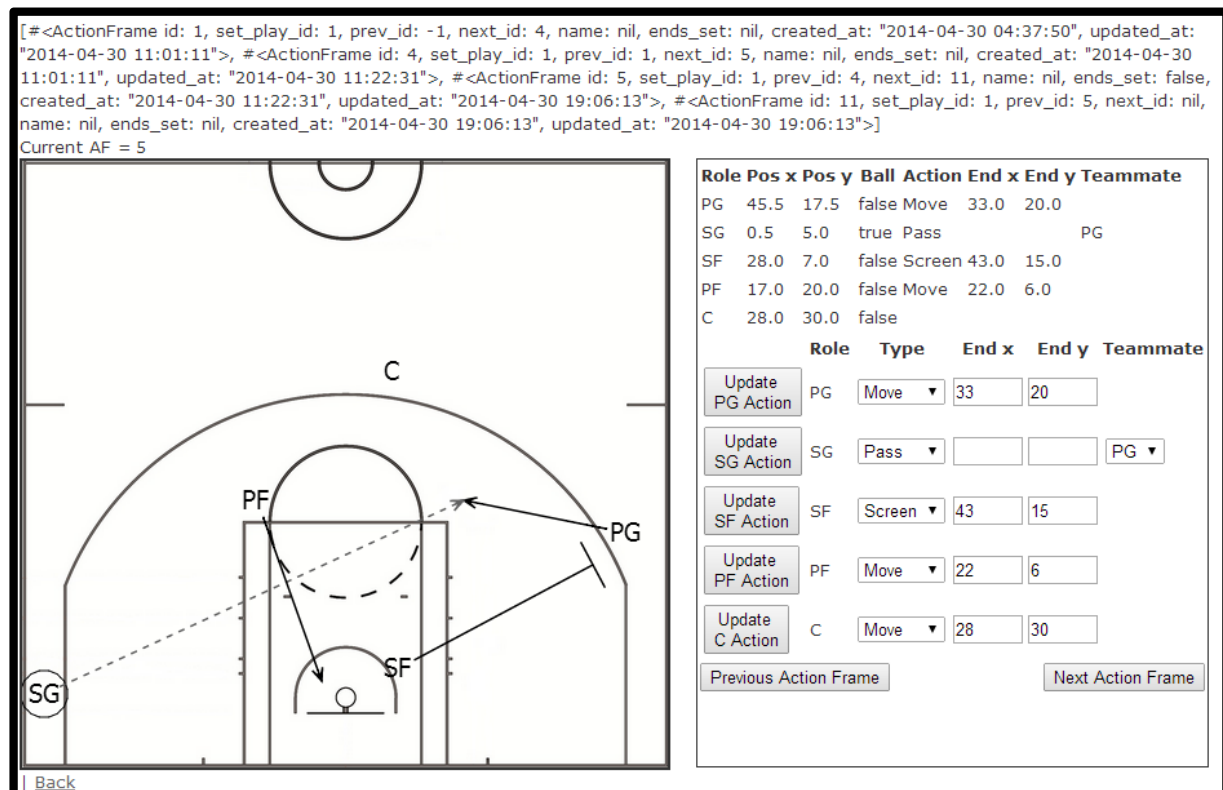


Figure 1 Screen and Pass to moving teammate

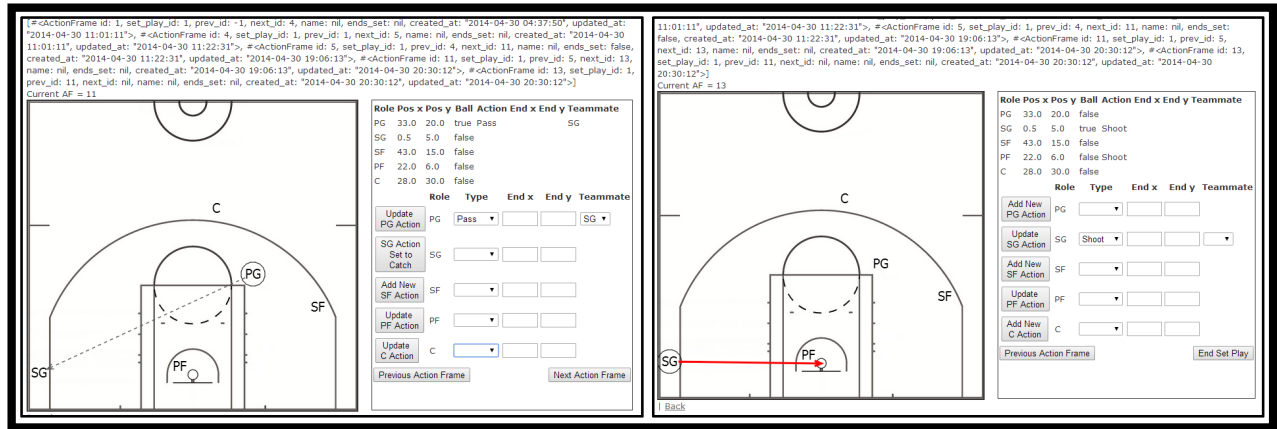


Figure 2 Pass to stationary teammate and Shot

Finally, the animation was possibly the most challenging part of the project. I had originally hoped to have two different animation methods, one for animating a single action frame and another for animating an entire set play from a given starting point to its end. Unfortunately, the set play animation required a large amount of data sharing between the Ruby on Rails framework and the Coffeescript code that dynamically rendered the Canvas HTML, and the two sections of the application can only communicate through Ruby-embedded Javascript. This is where Rails' lack of a graphics library had the largest and the most disappointing impact.

However, the single action frame animation was manageable as all the necessary data is collected by the Javascript while displaying the action lines. I added a button and binded a Coffeescript function to it. When the button is clicked, the Coffeescript clears the canvas and calculates the path along which all moving bodies will move. They are then displayed at an incremented distance down that path, then the canvas is cleared again and the process starts again until the destination is reached.⁵ The time increment had to be found by trial and error in order to keep halting at destination consistent as well as the speed of travel to be reasonable. 85 milliseconds was found to work well.

⁵ This is actually done in similar fashion to the drawing of dashed lines

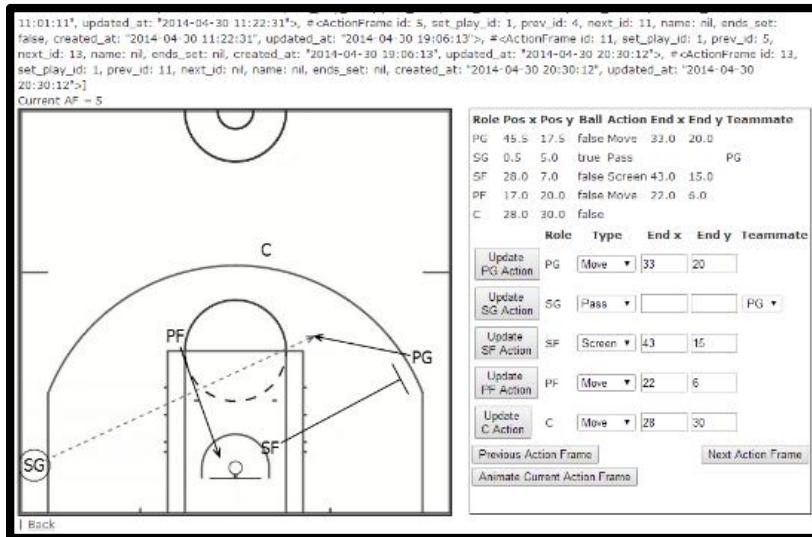


Figure 3 Before Animation

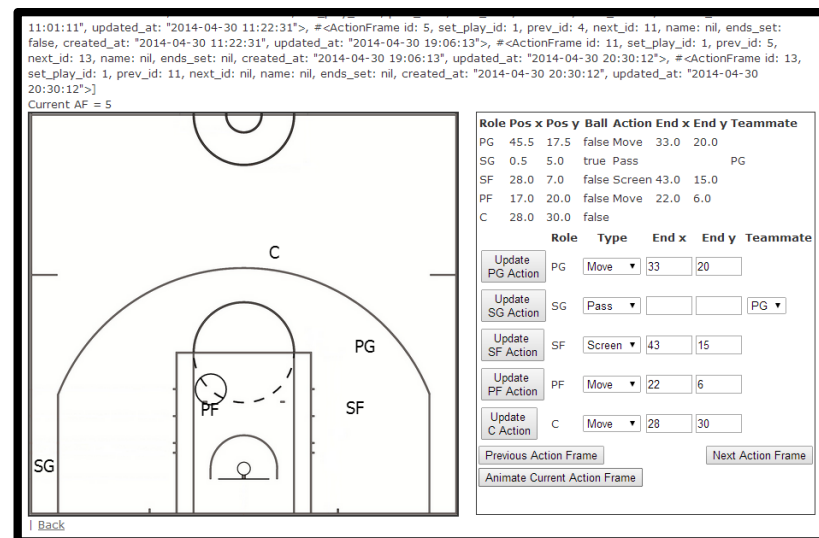


Figure 4 During Animation

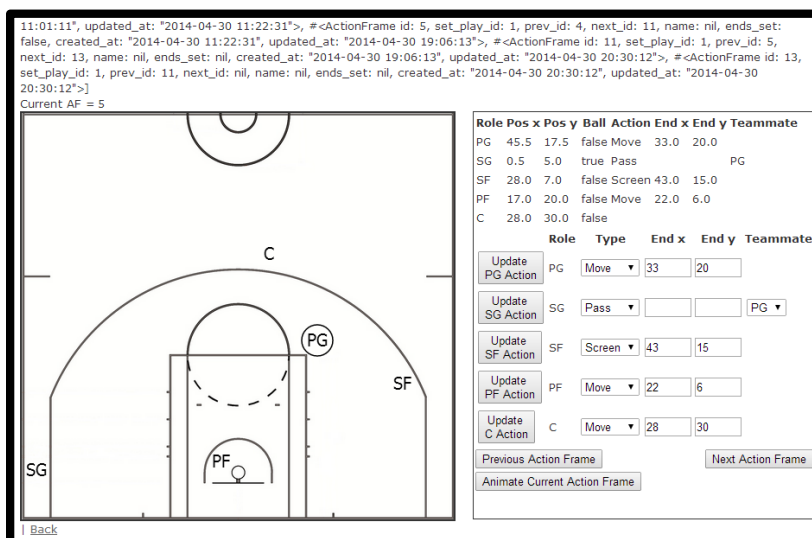


Figure 6 After Animation

Looking back, I learned a lot with the project. Firstly, my understanding of the Ruby on Rails framework is infinitely more comprehensive than it was when I first started. Having to go back and forth between HTML, embedded Ruby, non-embedded ruby, Javascript and Coffeescript was challenging, but also eye opening. Many times I wanted to pass data from one part of the application to another, either from the HTML view to the Javascript code, or from the HTML view to the Rails controller, and it just would not work. Finding a new way to do something seemingly impossible, even as simple as creating actions for an action frame, was a moment of success and growth. It is the kind of moment I was hoping for when I first came up with the idea of a whiteboard app. In truth, I thought this project would teach me primarily about graphics and GUIs, but choosing Ruby on Rails was not the way to learn about those things. Instead, I learned Ruby on Rails and how to build something up completely from scratch. I also finally learned how to dynamically call Javascript, jQuery and Coffeescript methods, something that had been befuddling me for months.

However, if I could go back and start again, I'd do things differently. I'm more than glad that I built what I did and learned what I did, but if I could I would start this project again in pure Java. I would use all the graphical display the language has to offer, and I would build a true-to-form application with the hopes of having the program tablet-ready by submission time. That being said, when I started this project I had a million and eight ideas on what functionality I wanted to implement and I was ready and willing to spend all my time on each one of them. I think that, had I gone straight to Java, I would not have established the set play of action frame of action framework that I did. And it is that framework that I now hope to build on, and perfect this project.

I'd like to thank Professor Holly Rushmeier, for putting up with me and seeing my through thick and thin. Her patience is unlimited and her support is unwavering.