The Dynamics of Real Time Competition: What Makes a Pro?

Adam Kunesh
UC Davis Dept. of Physics
apkunesh@ucdavis.edu
Final Project, Prof. Jim Crutchfield's PHY 256B
TuThu 12:10-1:30PM, SQ 2018

Abstract

Competitive video gaming is discussed using concepts developed in PHY 256. An experimental computational architecture, based loosely on Psyonix Studio's smash eSports hit, *Rocket League*, is used to explore relationships between process structure and game outcome. Ultimately, questions of the deeper meanings behind "competition" and "effective play" are left unanswered, but some tangential results indicate that concrete definitions could yet be produced.

Introduction

There's never been a better time to live as a professional video game player. Indeed, before the year 2000[1], playing video games for cash was, at best, a diverting second source of income. With the evolution of broadcast eSports (Electronic Sports) leagues and tournaments, truly obsessed and talented gamers gained a spotlight. Average Joes who played games were willing to pay to see the best; thus a profession was born. With the development of web-based video streaming services like Twitch[2] in 2011, droves of "pro gamers," mostly young and charismatic, now pay rent by entertaining the masses and selling fan merchandise.

What traits give these pro gamers their abilities to perform? Can anyone be a pro given enough training, or, as with professional athletics, do pro gamers have inherent physical advantages, like reaction time? What makes a game worthy of competition for its players? That is, what attracts gamers to certain kinds of games? Here, I provide a few insights into how we might go about answering these questions in the framework of machines.

Background and Questions

When I was young, I played games like Smash Bros 64, Halo: Reach, and a few of my dad's favorite "old" arcade games. When I wasn't blaming RNG (random number generation) for my losses, I convinced myself that I had been technically outperformed. That is, there was nothing I could've done to win; the person I was playing against had too much practice jiggling their thumbs in just the right way to beat me.

Then I met Rocket League, and the whole tone of video games changed. I'd never before played a competitive game at a competitive level. That is, I was never technically skilled enough to start playing mind games with my opponents, to attempt to break them not by being more mechanically adept but strategically adept. When I began searching for final project topics, I realized that machines and stochastic data generators could be a useful framework in which to attempt getting a deeper understanding of games and the many layers of competition. After all, competitive games consist of players who interact with a game through digital inputs, changing the dynamics of the game for the opponent.

Let's get concrete: it may be possible to codify a player's choice of state (as symbolized by an input or set of inputs on a controller) as a Markov chain with probabilities weighted by observations of the opponent's playstyle and the physics of the game (Figure 1). By playing the game, he could construct a set output-generating states which result in victories given inputs communicated by the game. Further, maybe a game can be reduced to a communication channel through which opponents push information about their own internal processing. Then a game would be not a test of technical ability alone (alphabet), but also of the ability to appropriately interpret the signals generated by the opponent (synchronization).

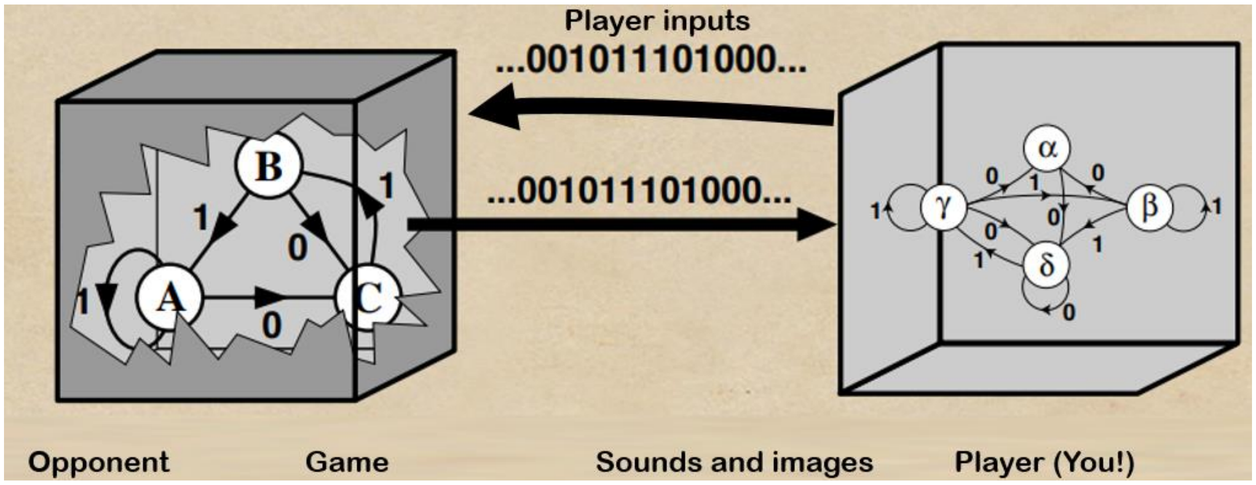

Figure 1: Modified "Modelling a Process" Diagram from PHY 256A Lec. 1[3]. Here, inputs are communicated between players through the game. Attempts to synchronize to the opponent allow for prediction of the opponent's future actions, allowing the player to establish counter-moves to produce more favorable outcomes.

Immediately this framework lends itself to a ton of questions. Among my favorite are:

1) How does the structure of a game's rules affect its level of competitiveness?
2) How does a game's structure affect the structure of a player's processing?
3) In a given game, how does an opponent's structure affect the required structure of the opponent?
4) What structure allows some pros to master multiple competitive games?
5) What limits do the time constraints of real-time play exert on playstyle?
6) Can we, in some sense, "separate" mechanical ability from strategic play?
7) Does successfully beating an equally mechanically skilled opponent amount to synchronizing to their processes and choosing paths which defeat those options?

While 1-5 are interesting and broadly applicable, I felt inadequately equipped to tackle these problems, which seem to have a close connection to emergence. I chose instead to focus on 6 and 7. Drawing inspiration from the Super Smash Bros. Melee community, I hoped to decompose good gameplay into two components: mechanical ability, which would correspond to a large number of possible states, and ability to synchronize to the opponent, for optimal prediction. On online forums, the former would be referred to as "tech" or "techskill" while the latter would be called "reads," shorthand for "reading your opponent."

Could I produce a simple enough competitive game to show that, at least for some games, a winning player was one capable of accessing all states and synchronizing to (or "reading") the opponent?

Experiment

I chose to give Python a shot for this project, as it seemed to be a choice for rapid prototyping. Also, I wanted to animate game simulations, and I already played around with MatPlotLib during PHY 256. Due to my familiarity with the game, I chose to drastically simplify and encode Rocket League. Rocket League may be most briefly described as "car soccer." In the

1v1 game mode, two players face off on a rectangular map, attempting to score goals at opposite ends of the pitch. (Many video examples of gameplay can be found on YouTube.)

First I developed rules for the arena. I chose to make positions for the two players and ball discrete points on a 430 by 350 grid. I gave the arena walls, and coded in initial positions for the three actors. I designed in ball logic less like soccer and more like football: when a player picks up the ball, they turn green to symbolize possession, and the ball travels with them. The two hardest decisions to make involved “contests,” where an opposing player attacks the player in possession of the ball, and player AI. For contests, I settled on slightly randomizing the position of the ball (fumbling) and also randomizing the positions of the players (bumping). After each contest, the players would be, on average, an equal distance from the ball.

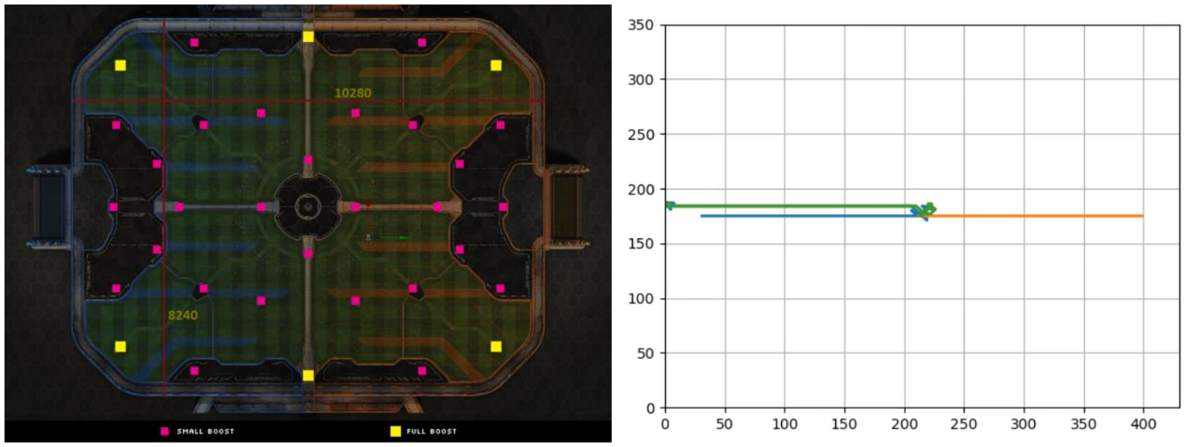

Figure 2: *Left* Rocket League Arena, top view (image courtesy Reddit user Psyonix_Dave). Note the two goals the left and right ends, along with the round ball in the center of the court. *Right* All positions during a simulated game between two perfect players. Orange wins!

Choosing what sorts of strategies to give the players was tricky, not only because I wanted to make the game interesting to watch, but also because I wanted to be able to generate the sorts of actions the player might take with machines like those we’d developed during the course. Because I wanted to break the game into two components, mechanics and strategy, I designed several player archetypes: undertrained vs. perfect mastery of controls, and random vs. optimal decision making. On the “mechanics” side, the undertrained player was allowed to move left and right, only, while the perfect player was allowed to move any of 8 directions on the grid (Figure 3). On the strategy side, the random player would choose his next direction randomly among those provided, while the optimal decision maker would move toward the ball as fast as possible and, when in possession of the ball, as fast as possible to the opponent’s net.

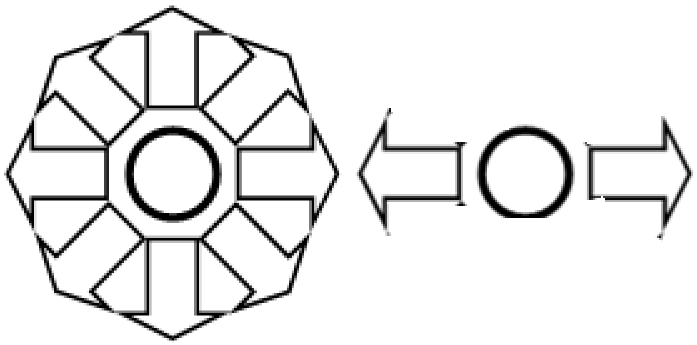

Figure 3: Allowed movement options for the Perfect Mastery (left) and Undertrained Mastery (right) mechanical archetypes.

With these archetypes in hand, I pitted them against each other in my 2-D Rocket League clone and observed the results.

Results

Perfect Mastery, Perfect Strategy vs Perfect Mastery, Perfect Strategy:

These two equally-well-equipped players would converge on the ball's location at the middle of the field and fight for a few contests before one would break away and score a goal. Who would score was a coinflip.

Undertrained Mastery, Perfect Strategy vs Perfect Mastery, Perfect Strategy:

The undertrained player's lack of inputs damaged his ability to react to the more mechanically adept perfect player, despite optimal positioning. After a single contest at midfield, the double-perfect player won nearly all the time, though, rarely, the undertrained player would get lucky during contest randomization and score a runaway goal.

Perfect Mastery, Random Strategy vs Perfect Mastery, Perfect Strategy:

In this game, random strategy was worse-than-useless when combined with perfect mastery. Typically, the perfect player would sweep the ball from midfield and charge directly into the meandering random player's net; on the rare occasion of a contest, the perfect player would simply re-challenge to gain control of the ball and score.

Perfect Mastery, Random Strategy vs Undertrained Mastery, Perfect Strategy:

When two flawed players battled, perfect strategy dominated over mechanical ability. In the rare event of a contest, the game would typically end in a stalemate (that is, the simulation would end before a scored goal).

The victory table below summarizes the results.

(Mastery, Strategy)	***(P,P)***	***(P,R)***	***(U,P)***	***(U,R)***
(P,P)	Coinflip	(P,P)	(P,P)	(P,P)
(P,R)	(P,P)	Stalemate/coinflip (untested)	(U,P)	Stalemate/(U,R) (untested)
(U,P)	(P,P)	(U,P)	Stalemate/coinflip (untested)	(U,P)
(U,R)	(P,P)	Stalemate/(U,R) (untested)	(U,P)	Stalemate/coinflip (untested)

Table 1: (P,P) here means "Perfect Mastery, Perfect Strategy;" U means "Undertrained Mastery," while R means "Random Strategy." The most common game outcome given the bolded competitors is given in each corresponding cell.

We can see that, for matchups that didn't take too long to simulate, perfect strategy is a common theme. Undertraining is only more likely to come out on top when both players' decision-making is randomly distributed over their options.

Discussion and Conclusion

Right off the bat, I'll say I failed to show that being "good" at a competitive game amounts to mechanical ability and synchronization to an opponent. However, I do think I made some interesting headway.

I failed to test my hypothesis because the game I designed wasn't competitive. The optimal strategy for the game I designed did not depend on the opponent's position as an input. There was no need to interface with an intelligent actor to win. Playing the game optimally required only the ball's position. Certainly, then, *a competitive game does require that opponents exchange information*. Further, there was only one optimal strategy, and it won an overwhelming majority of the time; there was no "rock" to the optimal "scissors," no weak points that would require adaptation on the part of the aggressor. *A competitive game requires balance* in the sense that there must be a variety of playstyle states which act as counters to other favorable playstyle states.

I also think I learned something about the utility of randomness in competitive play. In games with a large parameter space, like Rocket League or my drastically simplified clone, random inputs result in loss a huge percentage of the time. (From the initial position, even just holding in the direction of the opponent's goal has a better chance of winning.) In games with a much smaller set of options, though, like rock-paper-scissors, choosing perfectly randomly should yield roughly 50% wins regardless of your opponent's strategy.

One particularly interesting case in the victory table is the undertrained random strategist vs the master random strategist. The former can only move left and right, while the latter can also move up, down, and along diagonals. Because the ball and goals lie on a line from the players' initial positions, it stands to reason that the player with fewer options will eventually move to the center of the pitch, gain control of the ball, and defeat the player with more options, who will simply move out of the way. Attempting to generalize, a player who is randomly pressing only a few useful buttons has an advantage over a more technically skilled player who randomly explores his available motions. With lower entropy rate, number of states, and complexity, given the right circumstances, a player can still routinely beat a "more qualified" opponent. *It seems that there is likely not a 1-1 mapping between game structure and "ideal" entropy rates or statistical complexities on the part of the players.*

So what makes a pro? My guess is that an ideal player's process must have a high enough entropy rate to be difficult to synchronize with, but a low enough entropy rate to still have directed, efficient play. An ideal player should also be able to perform any action, but only when appropriate—overly technical play can actually be a handicap in some games.

Future Study

In addition to the unaddressed questions mentioned in the Background and Question, one which seemed increasingly relevant as I continued on this project was: according to a player viewing the outputs of a game, can the generating process be separated into "game rules" and "opponent strategies," or are the two hopelessly entangled? My guess is that, in general, they are inextricable. However, in the same way that the game I designed didn't require the opponent's position to have optimal strategy, some special cases may exist.

If I wanted to replicate this project, I think I would have a better idea of how to design a competitive game. It would need to be balanced such that several different strategies could outweigh others, and players could develop an understanding of the opponent's preferred playstyles to generate appropriate responses. Further, instead of coding player behaviors directly, I think I would like to generate inputs directly from a model, rather than attempting to produce a model in hard code. This would allow for simpler quantitative discussion, which was admittedly difficult.

If the relationships between the structures of the game's rules, the opponent's playstyle, and the player's required responses can be generalized, efficient design of optimal players could eventually be achieved. This could have impact far beyond the initial impetus of individual improvement in video games, into the realm of general learning and strategy.

Bibliography

1) “Esports: A Brief History,” Web. http://adanai.com/esports/
2) “Twitch: About,” Web. https://www.twitch.tv/p/about/
3) Jim Crutchfield, “Overview”. *PHY 256A*. Web. http://csc.ucdavis.edu/~chaos/courses/ncaso/Lectures/Lecture1Slides.pdf
4) C. R. Shalizi and J. P. Crutchfield, "Computational Mechanics: Pattern and Prediction, Structure and Simplicity", Journal of Statistical Physics **104** (2001). Web. http://csc.ucdavis.edu/~cmg/papers/cmppss.pdf

Appendix: Code

```python
import numpy as np
from scipy.spatial.distance import pdist, squareform
import matplotlib.patches as patches
import matplotlib.pyplot as plt
import scipy.integrate as integrate
import matplotlib.animation as animation
import random
import time

#control panel
matchLength=500
car1Behavior = 1  #1-->randomWalk, 2--> ballChase and score 3-->can't move up or down
car2Behavior = 3
ballBehavior = 1
challengeRadius = 10
animationName = "example.mp4"

#initial state
car1vec = np.array([[30,175]]) #should be 30, 175 and 400, 175; goals run from 70 to 140
car2vec = np.array([[400,175]])
ballvec = np.array([[215,175]])
car1possession = 0
car2possession = 0
challengeState = 0


#functions we'll need
    #high-position update functions
def updatecar1(car1xy, car2xy, ballxy):

    if car1xy[0]<0 or car1xy[0]>430 or car1xy[1]<0 or car1xy[1] > 350:
        car1new = correctOutOfBounds(car1xy)
    elif car1Behavior==1:
        car1new = randomWalk(car1xy)
    elif car1Behavior == 2:
        if car1possession==0:
            car1new = ballchaseMode(car1xy, ballxy)
        else:
            car1new = straightToRightGoal(car1xy)
    else:
        print('error in updatecar1')
    return car1new
def updatecar2(car1xy, car2xy, ballxy):
    if car2xy[0]<0 or car2xy[0]>430 or car2xy[1] < 0 or car2xy[1] > 350:
        car2new = correctOutOfBounds(car2xy)
    elif car2Behavior==1:
        car2new = randomWalk(car2xy)
    elif car2Behavior == 2:
        car2new = ballchaseMode(car2xy,ballxy)
        if car2possession == 0:
            car2new = ballchaseMode(car2xy, ballxy)
        else:
            car2new = straightToLeftGoal(car2xy)
    elif car2Behavior == 3:
        car2new = noUpOrDown(car2xy,ballxy)
        if car2possession == 0:
            car2new = noUpOrDown(car2xy,ballxy)
        else:
            car2new = moveLeft(car2xy)
    else:
        print('error in updatecar2')
    return car2new
def updateball(car1xy, car2xy, ballxy):
#In this version, the ball doesn't move unless possessed by a player. Then in follows the
player's movement.
    if ballBehavior ==1:
        if car1possession==1 and car2possession==1:
            ballxynew=[ballxy[0]+random.randint(-5,5),ballxy[1]+random.randint(-5,5)]
        elif car1possession ==1:
            ballxynew = [car1xy[0],car1xy[1]]
        elif car2possession == 1:
            ballxynew = [car2xy[0],car2xy[1]]
        else:
            ballxynew = [ballxy[0], ballxy[1]]
        return ballxynew
    else:
        ballxynew = [ballxy[0], ballxy[1]]

#low-level position update functions
def moveRight(thingxy):
    newthingxy = [thingxy[0]+1,thingxy[1]]
    return newthingxy
def moveLeft(thingxy):
    newthingxy = [thingxy[0]-1,thingxy[1]]
    return newthingxy
def moveUp(thingxy):
    newthingxy = [thingxy[0],thingxy[1]+1]
    return newthingxy
def moveDown(thingxy):
    newthingxy = [thingxy[0],thingxy[1]-1]
    return newthingxy
def moveUpRight(thingxy):
    newthingxy = moveRight(moveUp(thingxy))
    return newthingxy
def moveDownRight(thingxy):
    newthingxy = moveRight(moveDown(thingxy))
    return newthingxy
def moveUpLeft(thingxy):
    newthingxy = moveLeft(moveUp(thingxy))
    return newthingxy
def moveDownLeft(thingxy):
    newthingxy = moveLeft(moveDown(thingxy))
    return newthingxy
def correctOutOfBounds(objxy):
    if objxy[0] <= 0:
        newxy = moveRight(objxy)
    elif objxy[0] >= 430:
        newxy = moveLeft(objxy)
    elif objxy[1] <= 0:
        newxy = moveUp(objxy)
    elif objxy[1] >= 350:
        newxy = moveDown(objxy)
    else:
        print('ERROR IN correctOutOfBoundsWalk')
        newxy = objxy
    return newxy
def straightToLeftGoal(objxy):
    if objxy[1]<75:
        newobjxy = moveUpLeft(objxy)
    elif objxy[1]<135:
        newobjxy = moveDownLeft(objxy)
    else:
        newobjxy = moveLeft(objxy)
    return newobjxy
def straightToRightGoal(objxy):
    if objxy[1] < 75:
        newobjxy = moveUpRight(objxy)
    elif objxy[1] < 135:
        newobjxy = moveDownRight(objxy)
    else:
        newobjxy = moveRight(objxy)
    return newobjxy
def checkChallenge(x1,y1,x2,y2):
    if (x2-x1)**2+(y2-y1)**2<challengeRadius:
        trans=1
    else:
        trans=0
    return trans
```

```python
#individual "Car brains" or strategic "states"
def randomWalk(objxy):
    newxy = [objxy[0]+random.randint(-1,1),objxy[1]+random.randint(-1,1)]
    return newxy
def ballchaseMode(carxy,ballxy):
    relx,rely = relativePosition(carxy,ballxy)
    if relx>0:
        if rely<0:
            newcarxy = moveDownRight(carxy)
        elif rely>0:
            newcarxy = moveUpRight(carxy)
        else:
            newcarxy = moveRight(carxy)
    elif relx<0:
        if rely>0:
            newcarxy = moveUpLeft(carxy)
        elif rely<0:
            newcarxy = moveDownLeft(carxy)
        else:
            newcarxy = moveLeft(carxy)
    else:
        if rely>0:
            newcarxy = moveUp(carxy)
        elif rely<0:
            newcarxy = moveDown(carxy)
        else:#activate possession?
            newcarxy = carxy
    return newcarxy
def noUpOrDown(carxy,ballxy):
    relx,rely = relativePosition(carxy,ballxy)
    if relx>0:
        if rely<0:
            newcarxy = moveRight(carxy)
        elif rely>0:
            newcarxy = moveRight(carxy)
        else:
            newcarxy = moveRight(carxy)
    elif relx<0:
        if rely>0:
            newcarxy = moveLeft(carxy)
        elif rely<0:
            newcarxy = moveLeft(carxy)
        else:
            newcarxy = moveLeft(carxy)
    else:
        if rely>0:
            newcarxy = moveLeft(carxy)
        elif rely<0:
            newcarxy = moveLeft(carxy)
        else:#activate possession?
            newcarxy = carxy
    return newcarxy

#navigation
def relativePosition(obj1,obj2):
    xpos,ypos = obj2[0]-obj1[0],obj2[1]-obj1[1]
    return xpos, ypos

#unused
def plotState(car1xy, car2xy, ballxy):
    plt.plot(car1xy[0],car1xy[1],'ro')
    plt.plot(car2xy[0],car2xy[1],'bo')
    plt.plot(ballxy[0],ballxy[1],'ko')
    plt.show()

#generate the dataset (MAIN RULES OF THE GAME!!!)
for x in range (0,matchLength):
    if car1vec[x][0]==ballvec[x][0] and car1vec[x][1]==ballvec[x][1]:
        car1possession = 1
    elif car2vec[x][0]==ballvec[x][0] and car2vec[x][1]==ballvec[x][1]:
        car2possession = 1
    else:
        car1possession = 0
        car2possession = 0

    challengeState = checkChallenge(car1vec[x][0],car1vec[x][1],car2vec[x][0],car2vec[x][1])

    if challengeState==1:
        dummy = [car1vec[x][0]+random.randint(-10,10),car1vec[x][1]+random.randint(-10,10)]
#defines rules of 50-50s
        car1vec=np.vstack((car1vec,dummy))
        dummy = [car2vec[x][0]+random.randint(-10,10),car2vec[x][1]+random.randint(-10,10)]
#defines rules of 50-50s
        car2vec=np.vstack((car2vec,dummy))
    else:
        dummy=updatecar1(car1vec[x][:],car2vec[x][:],ballvec[x][:])
        car1vec=np.vstack((car1vec,dummy))
        dummy=updatecar2(car1vec[x][:],car2vec[x][:],ballvec[x][:])
        car2vec = np.vstack((car2vec, dummy))

    dummy=updateball(car1vec[x+1][:],car2vec[x+1][:],ballvec[x][:])
    ballvec = np.vstack((ballvec, dummy))

#-----------------------------------------BIG DIVIDE HERE------------------------------------
#animate the dataset
def data_gen():
    t = data_gen.t
    cnt = 0
    while cnt < matchLength:
        cnt+=1
        t += 0.01
        yield car1vec[cnt][0], car1vec[cnt][1], car2vec[cnt][0], car2vec[cnt][1],
ballvec[cnt][0], ballvec[cnt][1]

data_gen.t = 0
fig, ax = plt.subplots()
line, = ax.plot([], [], lw=2)
line2, = ax.plot([], [], lw=2)
line3, = ax.plot([], [], lw=2)
ax.set_ylim(0, 350)
ax.set_xlim(0, 430)
ax.grid()
xdata, ydata = [], []
x2data, y2data = [],[]
x3data, y3data = [],[]
def run(data):
    # update the data
    t,y, t2, y2, t3, y3 = data
    xdata.append(t)
    ydata.append(y)
    x2data.append(t2)
    y2data.append(y2)
    x3data.append(t3)
    y3data.append(y3)

    ax.figure.canvas.draw()
    line.set_data(xdata, ydata)
    line2.set_data(x2data, y2data)
    line3.set_data(x3data, y3data)

    return line3, line2, line,


ani = animation.FuncAnimation(fig, run, data_gen, blit=True, interval=40,
    repeat=True, save_count = 500)

ani.save(animationName)

plt.show()
```