

Learning to Bounce a Ball in Virtual Reality

Final Project

Team Members:

Sadra Hemmati, Brett Brownlee

Fall 2015

University of Tennessee, Knoxville

Outline

- I. Abstract
- II. Introduction and Background
 - A. What is VR?
 - 1. Serious Gaming in rehabilitation
 - 2. Use of Reinforcement Learning in VR for AI design
 - B. Problem Definition
- III. Design
 - 1. Design Objectives
 - a) Virtual Reality 3D implementation
 - (1) Networking to MM
 - (2) Physics Programming
 - b) Defining State Space
 - c) Defining Action Space
 - 2. Design Challenges
 - a) Real Player Recorded Data
 - b) Converting MATLAB to Unity
 - 3. Technical Approach
 - a) Eligibility Traces SARSA(λ)
 - b) Q-Learning
 - 4. Experiments and Results

- a) MATLAB simulations and testing
- b) VR system AI implementation

IV. Summary

V. Future Work

VI. Appendix A (Source code listing)

- A. simulation.m (Page 30)
- B. disc.m (Page 32)
- C. qlearning.m (Page 35)
- D. matlabToUnity.m (Page 38)
- E. physics.cs (Page 39)

Abstract

Reinforcement Learning is an active research area which studies the methodological approaches for enabling machines to perform intelligent and expert behaviours. Frameworks of Reinforcement Learning has been utilized for solving various types of problems including games such as Backgammon, Chess or control problems such as inverted pendulum. Here we implement two RL approaches namely Q-Learning and Eligibility Traces for enabling an agent to control the trajectory of a ball with a paddle. First, the game environment is defined as a discrete space and then the aforementioned algorithms are implemented using the MATLAB software. The results indicate that the agent learns in a reasonably fast time and is able to play

the game and keep the ball in the air for a considerable number of hits. The results of this project can be compared with human performance in order to infer physiological parameters affecting human learning.

Introduction

Virtual Reality (VR) holds promise in revolutionizing the rehabilitation field and medical physiotherapy. A sufficiently immersive VR system can be a valuable asset for many application domains. Previous work in the field spans the flight simulation training, driving simulation for specialized target populations and rehabilitative VR used for capacitating post-stroke patients and children with Autism. Due to its novelty and great promise in the healthcare domain we are interested in exploring the potential benefits of using VR in the upper body rehabilitation schemes where a healthy or injury-affected individual can play with an agent capable of adapting its behaviour to that of the human participant. The injury can be any type of movement disorders directly reducing upper limb functionality or neuromuscular diseases such as stroke or Parkinson's disease which both affect the decision making capabilities and anatomical dexterity of the individual. A very engaging paradigm for rehabilitation purposes for the aforementioned target populations is serious gaming. In this scheme, a well-designed game is utilized with certain dosages and adaptable levels to trigger body neuro-rehabilitative mechanisms to improve the patient condition. It is important to note that this stage of recovery comes after the acute stage where the patient is in direct supervision of a clinician and unable to fulfill the minimum requirements of playing a game such as alertness or vulnerability to participate in such an activity which could impose risks which are to be avoided.

Many of the current VR systems employ physics and game engines to simulate a scenario. One popular and efficient engine is Unity which has been used for developing games for PC, consoles, mobile devices and the web. Due to its increased platform independence compared to other engines, Unity offers game developers a great flexibility in choosing the programming architecture and language. In this project a C# based program was developed on Unity that implements a challenging motor learning task.

Use of RL algorithms is hypothesized to bring multiple benefits for the purpose of rehabilitation. Integrating Reinforcement Learning (RL) techniques and methods in the VR domain can help for studying human-machine interaction by creating AI agents to assist or challenge the player. Additionally, RL learning can give insight into designing systems rewards and parameters by exploring the entire state and action space.

This project implements the first step in this process. We implemented a challenging virtual motor task that a non-disabled player can improve at and also created a virtual agent to do the same. Because of the nature of virtual reality, all of the parameters of the task and its difficulty have the potential to be adjusted to meet anyone's skill and ability level. We aim to use the observations of the player and agents learning and actions to minimize the threshold for learning to play our task.

Problem Definition

In order to take full advantage of VR's ability to enable learning, a problem was defined that is sufficiently challenging for a person to solve optimally that also allows for difficulty scaling with virtualization. We decided to model the game after the 2010 real life Hasbro game Bop It! Bounce.



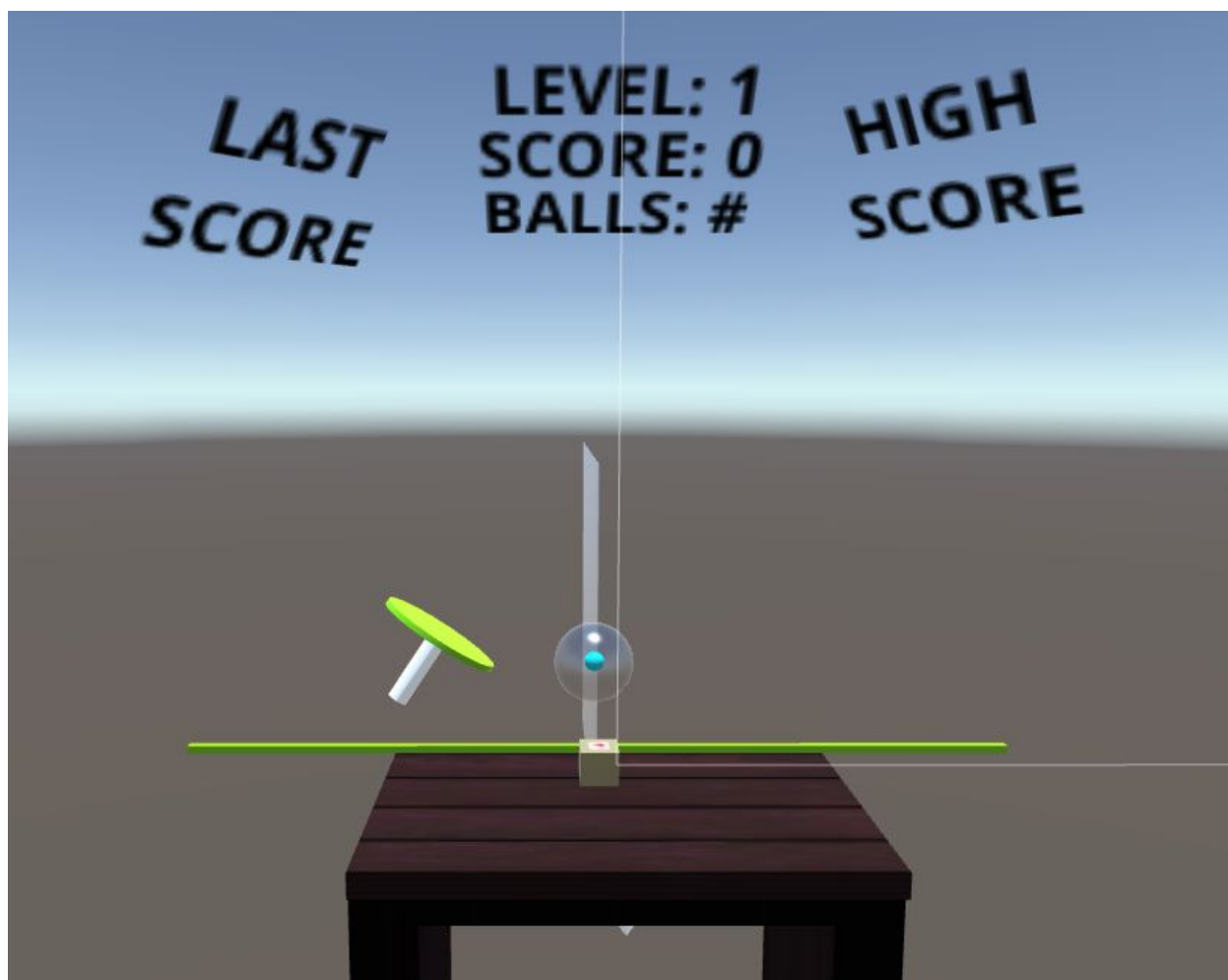
Bop It! Bounce (Figure 1)

Bop It! Bounce is an electronically equipped paddle with an elastic bouncing surface and a foam ball. By detecting the ball's impact, the software has a rough idea of how the player is bouncing the ball and if the ball is dropped. With this information, there are 6 different game settings that score the player for different bounce techniques such as number of bounces in a row, number of bounces in a set duration of time, maximum number of seconds in the air, and number of bounces that last for a set duration.

In real life, this game is fast paced and challenging. Novice players constantly hit the ball out of their reach due to sporadic hit patterns while experts have more stable trajectories. Due to people's various arm trajectories and dexterities, the problem generalizes to this: Once I get to the ball, how can I and how should I hit it to not drop it? Each hit is a combination of determined paddle angle and velocity. Due to the natural movement of the motion and the speed of the game, this question is answered almost unconsciously. The problem we ask our agent to learn is the same. Given a ball trajectory and location, determine the optimal paddle angle and velocity so the ball will not be dropped.

Design

The first step in our design was to implement the bouncing game into virtual reality. This was done in Unity with an Oculus Rift Dev Kit 1 head mounted display and The Motion Monitor magnetic positional trackers.



Virtual Game Implementation (Figure 2)

After putting on the headset, the player uses a floating paddle to pop the bubble and begin bouncing the ball. When the ball is dropped, the ball is reset back to its starting bubble. As the player bounces, their score increases and after set milestones so does the level. The timescale of the simulation increases with each level to slowly increase difficulty. The physics and tracker networking code can be seen in appendix A.

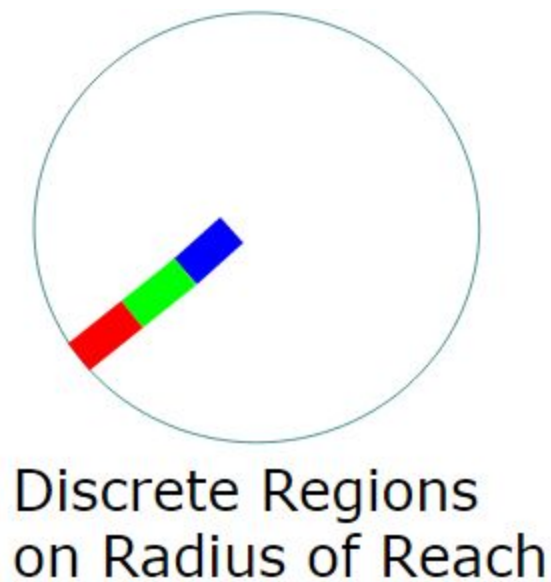
Once we proved that the game was playable in VR, we began designing how to simplify it to run simulations. We decided to implement the learning separately in Matlab. Our goal was to

design a state space with less than 100,000 possible combinations of states and actions. To do this, we first simplified the states down to its core components.

Where is the ball in my range of reach?

How is the ball moving?

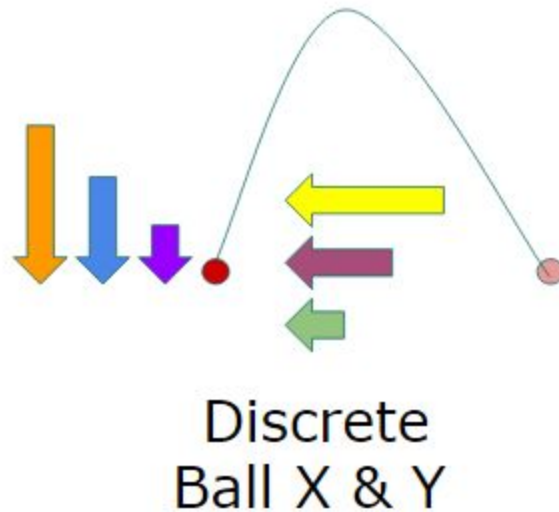
Reach was defined as a circle.



(Figure 3)

Reach is defined by the range the arm/paddle can move. The arm fully extended or retracted are the limits of the paddle's reach. The center of reach is the point in the middle of these maximum values. This mirrors the state space along two dimensions where there are regions that are more or less out of reach. For our initial design, we defined 5 discrete regions over a .5 meter range of reach.

Once the positions were defined on a linear 2D axis, the ball impact trajectory was simplified to the velocities on this axis.



(Figure 4)

Along this axis, 11 ball X velocities were initially designed as being $\pm(0\ 1\ 2\ 3\ 4\ 5)$ m/s. and 6 Y velocities $(0\ -1\ -2\ -3\ -4\ -5)$ m/s.. This state simplification is sufficient if the ball is only being bounced in two dimensions across the X axis of the players reach. For basic 3 dimensionality, it can be easily duplicated perpendicularly onto the Z axis.

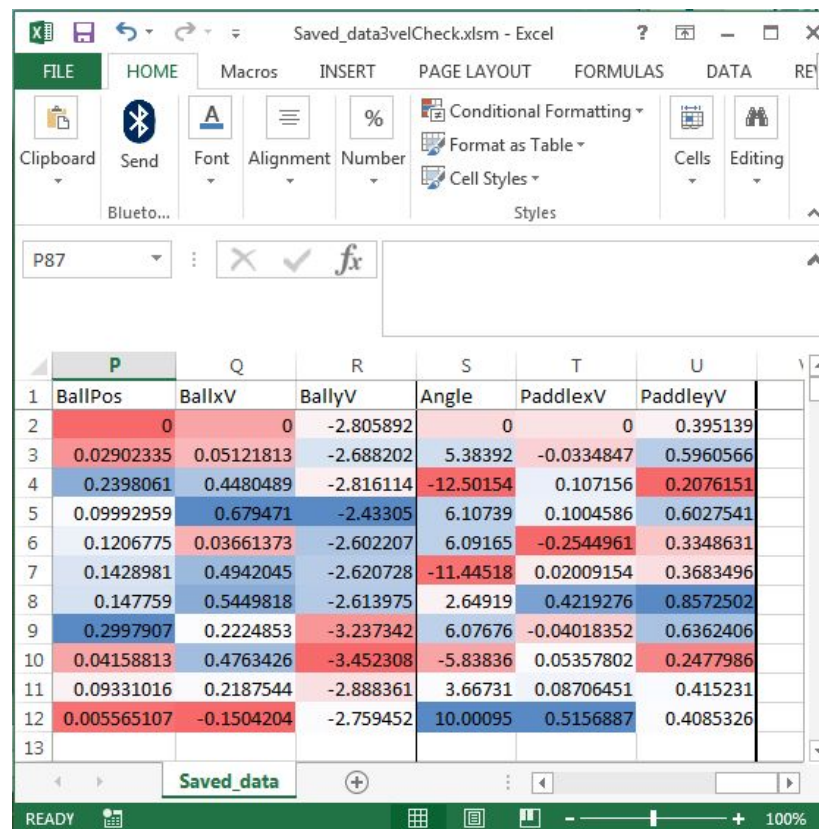
The action space was designed similarly. The 3 actions are determining paddle angle and paddle X&Y velocity. These were initially defined along the ball position axis with degrees of $(0\ 10\ 20\ 30\ 40)$ and velocities of $(-2\ -1\ 0\ 1\ 2)$ m/s.

Design Challenges

Designing Good Actions:

One issue with the above design is determining what actions to put into the discrete action space. The values chosen above were to cover a wide range of possible inputs. However, these might not be the most robust or natural.

One answer to this that has additional added benefits is to determine actions based on recorded human data. To accomplish this, code was added to the VR simulation to output all the actions a human player takes and the states that get visited.



	P	Q	R	S	T	U
	BallPos	BallxV	BallyV	Angle	PaddlexV	PaddleyV
1						
2	0	0	-2.805892	0	0	0.395139
3	0.02902335	0.05121813	-2.688202	5.38392	-0.0334847	0.5960566
4	0.2398061	0.4480489	-2.816114	-12.50154	0.107156	0.2076151
5	0.09992959	0.679471	-2.43305	6.10739	0.1004586	0.6027541
6	0.1206775	0.03661373	-2.602207	6.09165	-0.2544961	0.3348631
7	0.1428981	0.4942045	-2.620728	-11.44518	0.02009154	0.3683496
8	0.147759	0.5449818	-2.613975	2.64919	0.4219276	0.8572502
9	0.2997907	0.2224853	-3.237342	6.07676	-0.04018352	0.6362406
10	0.04158813	0.4763426	-3.452308	-5.83836	0.05357802	0.2477986
11	0.09331016	0.2187544	-2.888361	3.66731	0.08706451	0.415231
12	0.005565107	-0.1504204	-2.759452	10.00095	0.5156887	0.4085326
13						

Recorded Player State Data (Figure 5)

By recording lots of player hits, an idea of a natural state space can be defined from reoccurring values. By observing player behavior, the following state and action space was defined:

States:

Ball distance from center: (0, .1, .2, .3, .4, .5) meters

Ball x velocity: (-1-, -.9, -.8, -.7, -.6, -.5, -.4, -.3, -.2, -.1, 0, .1, .2, .3, .4, .5, .6, .7, .8, .9, 1+)

Ball y velocity: (0, -.5, -1, -1.5, -2, -2.5+)

Actions:

Paddle Angles: (0, 3, 7, 10, 30)

Ball x velocity: (-.3, -.1, 0, .1, .3)

Ball y velocity: (0, .1, .15, .3, 1)

Converting MATLAB to Unity:

Due to the data storage structures defined in Unity do not accept raw MATLAB multi-dimensional Q as input, (which is in line with C# data structures) , importing the multi-dimensional Q into Unity was a minor challenge that we had to address. Two pathways were considered in that matter: First was to utilize a proper address assignment formula which could enable the storage of the whole data in MATLAB Q matrix into a bulky two-dimensional matrix in the text format. This data type transfer can make the data transfer to Unity possible. Second way was to choose the optimal actions within each state and discard the remaining of the data in the Q.

The first approach states that for a matrix of $Q(M,N,Q,R,S,L)$ in MATLAB with the dimensionalities of M, P, ...,L , a straightforward formula to assign an address for each of the

$Q(i,j,k,m,n,p)$ to a single element in K (which is a large array of all numbers contained in the Q) can be in the form illustrated in the Eq.1 :

$$K(i+M*(j-1)+M*N*(k-1)+M*N*Q*(m-1)+M*N*Q*R*(n-1)+M*N*Q*R*S*(p-1))=Q(i,j,k,m,n,p)$$

Equation (1)

After construction of K , a search is needed to perform the reverse of the addressing, that is to find the proper indices from a single number. This process is a problem in number theory and is very interesting to tackle and easy to solve. The solution is based on the fact that multiple successive divisions can be formed and the indices of i,j,k,m,n and p easily extracted from that algorithm. Although the algorithm requires that the dimensions (M, P, \dots, L) be sorted in an increasing fashion such that $M < N < Q < R < S < L$ including the equality relationship considered.

Technical Approach:

In a parallel path to the Unity program development, a number of MATLAB codes were developed in order to perform the most critical part of the project which was implementation of a RL algorithm. Two approaches that will be discussed are as the following;

Eligibility Traces **SARSA**(λ) :

The first method for attacking the problem at hand was to use Eligibility Traces algorithm. The algorithm is shown in the Fig. 5. The premise with this approach was that the agent can find the optimal path in the multi-dimensional maze that we created for it (the total number of blocks that the agent could travel or the number of states is $7 \times 11 \times 11$ so the dimensionality of the maze is $7 \times 11 \times 11$). Use of eligibility traces can help in marking a sign of significance to the path that led to a successful outcome that is getting the ball inside the allowed partition. The parameters of learning were (which are reported in the code appendix)chosen based on the experience we gathered from the previous project (project number 2) and the reported numbers are not optimized from this perspective since more time was needed. The result of this implementation is a Q matrix which we could get the optimal actions from it at each state possible based on the experience gathered by the agent up to that point. The characteristics of this learning algorithm is as following:

First, consideration of a trajectory as the element of a ball bouncing event rather than a single state. This view is in line with the hypothesis that what the agent will do at state A is dependent on all the previous states visited at that episode of learning. The justifiability of this view is seen from the fact that some state-actions are way better than others (or the opposite, some state actions are much worse than the others) and good state-actions lead the agent to the other good state-actions so it is sensible to record the trajectory instead of the a single “good” state.

Second, the long run time of this algorithm which was due to the multitude of the conditional if-statements in the program.

```

Initialize  $Q(s, a)$  arbitrarily, for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ 
Repeat (for each episode):
   $E(s, a) = 0$ , for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ 
  Initialize  $S, A$ 
  Repeat (for each step of episode):
    Take action  $A$ , observe  $R, S'$ 
    Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
     $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$ 
     $E(S, A) \leftarrow E(S, A) + 1$  (accumulating traces)
    or  $E(S, A) \leftarrow (1 - \alpha)E(S, A) + 1$  (dutch traces)
    or  $E(S, A) \leftarrow 1$  (replacing traces)
    For all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ :
       $Q(s, a) \leftarrow Q(s, a) + \alpha \delta E(s, a)$ 
       $E(s, a) \leftarrow \gamma \lambda E(s, a)$ 
     $S \leftarrow S'; A \leftarrow A'$ 
  until  $S$  is terminal

```

The Eligibility Traces Algorithm (Figure 6)

However this method was not tried more than once due to its heavy computational load for MATLAB and significant amount of time which took for it to finish and give useful results. So we decided to continue further analysis with the more computer-friendly algorithm of Q-Learning.

Q-Learning:

Another approach taken was Q-Learning based approach and its algorithm is shown in the Figure 6. We implemented this scheme in MATLAB software. Essentially the goal of this process is to obtain a Q matrix of size(5,5,5,7,11,7) which is obtained by having the simulation of the events running and record all possible outcomes that could happen based on the all state-action space. Then this information was used to extract the optimal action at each state which is the policy improvement essentially. In order to provide the agent with enough exploration, at the beginning of each episode the state was chosen based on a random selection.

The total number of runs was over a million which well exceeds the state-space dimensionality ($7*11*7=539$) and confirms that enough exploration was considered in this fashion.

```

Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$ 
Repeat (for each episode):
  Initialize  $S$ 
  Repeat (for each step of episode):
    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
    Take action  $A$ , observe  $R, S'$ 
     $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$ 
     $S \leftarrow S'$ 
  until  $S$  is terminal

```

The Q-learning algorithm (Figure 7)

The MATLAB code was structured into three scripts of simulation.m, disc.m and qlearning.m.

- 1) simulation.m: This script calculates the behaviour of a ball based on classical mechanics.

Also this is the code wherein the rewards are assigned to different actions. Three types of rewards were designed which looked at different aspects of an ideal ball bouncing. The first and foremost is the reward for a successful catch of the ball. Another one is the reinforcement of keeping the ball in the air for a longer amount of time. The premise here was that an expert domain (a person who can play this game very efficiently) will try to keep the ball in the air for a longer period of time in order to provide enough reaction time and adding more control over the ball. The third reward mechanism was to give rewards based on the distance from the center. This is motivated by the fact that we

wanted the agent to keep the ball in the center of the limited space which mimics the behaviour of a person performing the task.

Experiments and Results:

MATLAB Results:

For each of the two taken approaches, a considerable amount of thinking was spent in order to contextualize the obtained results and infer precisely the behaviour of the agent from the output of the code which is essentially the Q matrix. The performed procedures and their related results are seen below:

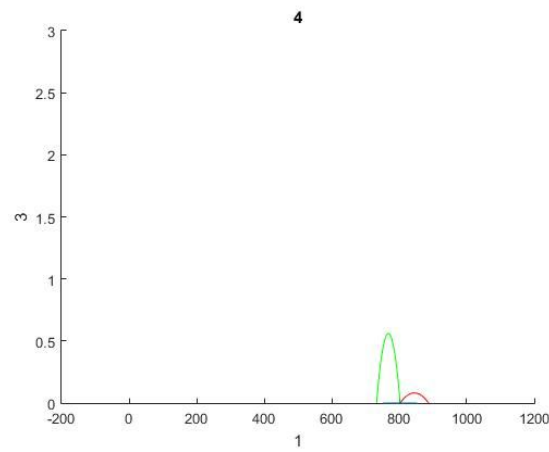
1) Comparison of run time of TD SARSA learning and Q-Learning:

One point worth attention is the difference between these two algorithms. From a FLOP size perspective, there is a huge difference between the two algorithms which showed itself in the run time very clearly. For example in a 2 minute time span recorded with a stop watch, the number of episodes finished in TD(λ) was two orders of magnitude smaller than the Q learning (232 for TD(λ) versus 10000 for Q-Learning).

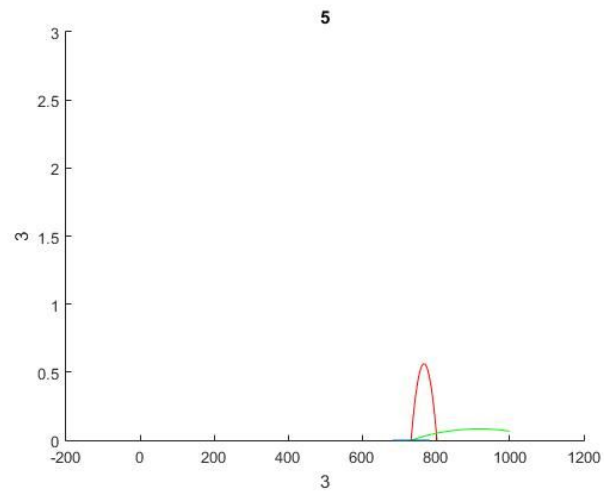
2) Plotting the trajectory of the ball within every episode:

To better visualize the simulation code a simple plot of ball trajectory was coded which helped to a considerable extent for that purpose. In the figures that follows, the red trajectory is the incoming ball and the green trajectory is the after-impact trajectory of the ball. Comparing two consecutive episodes, it should be the case that the green trajectory of the earlier snapshot be

the same as the red trajectory of the latter snapshot. The actions of the agent is not included graphically however the agent has a discoverable and extractable value for paddle angle and x and y velocities at each incidence of hitting the ball.



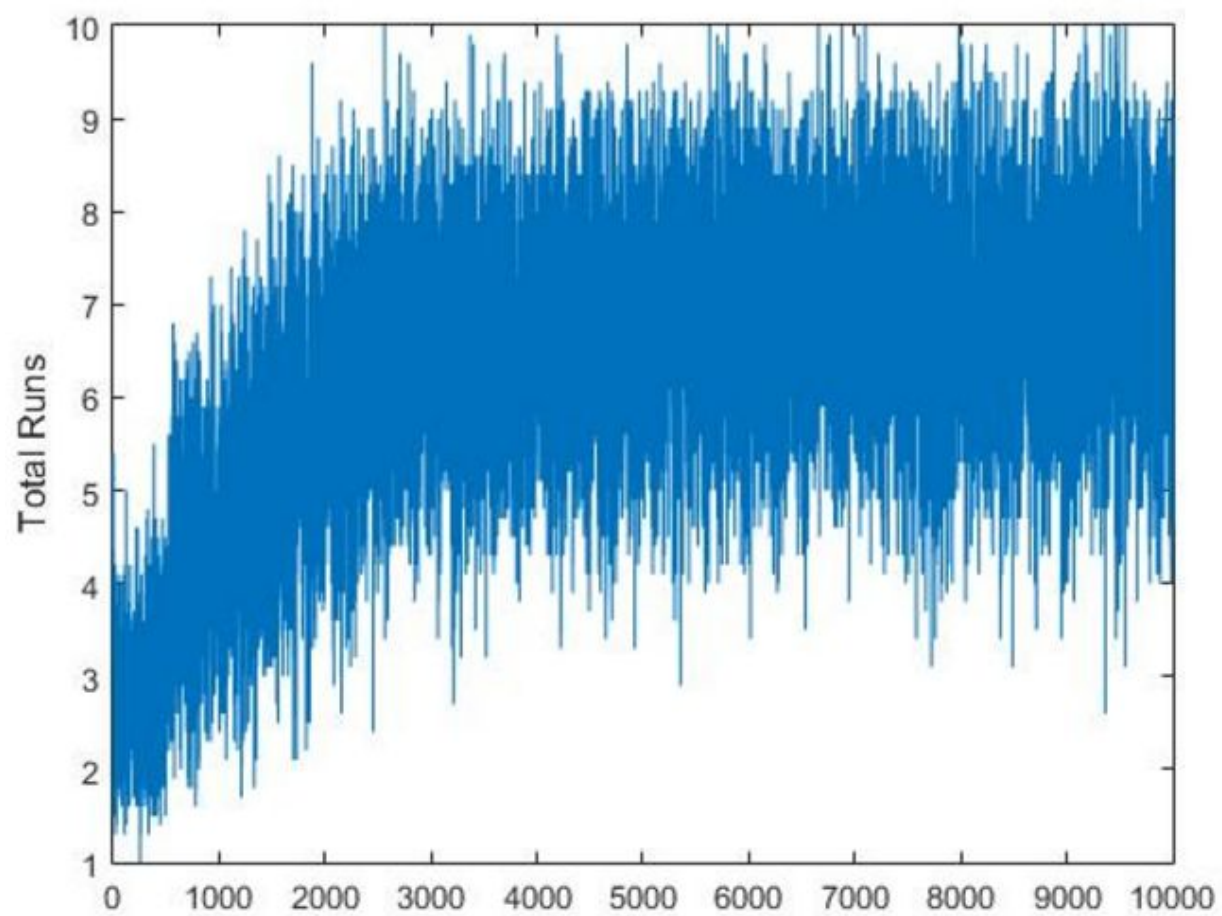
Ball Bounce Trajectories 1 (Figure 8)



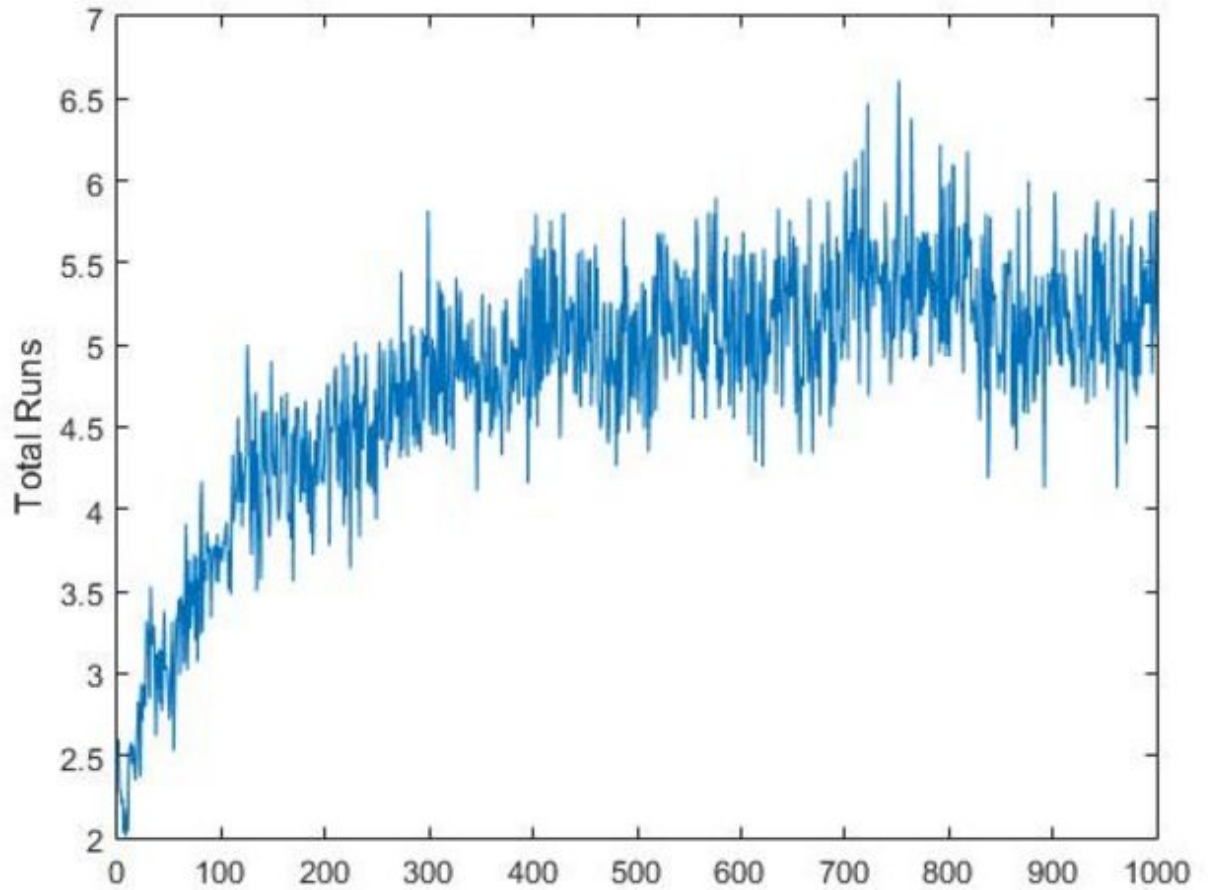
Ball Bounce Trajectories 2 (Figure 9)

3) Plotting the number of hits that the agent successfully completed at each episode

In another running of the qlearning script, the following graph was obtained as a result. The figure below indicates that for episodes of the learning process, the agent's ability will increase in hitting the ball consecutively. This image demonstrates the improvement trend obtained by taking the Q-Learning approach.

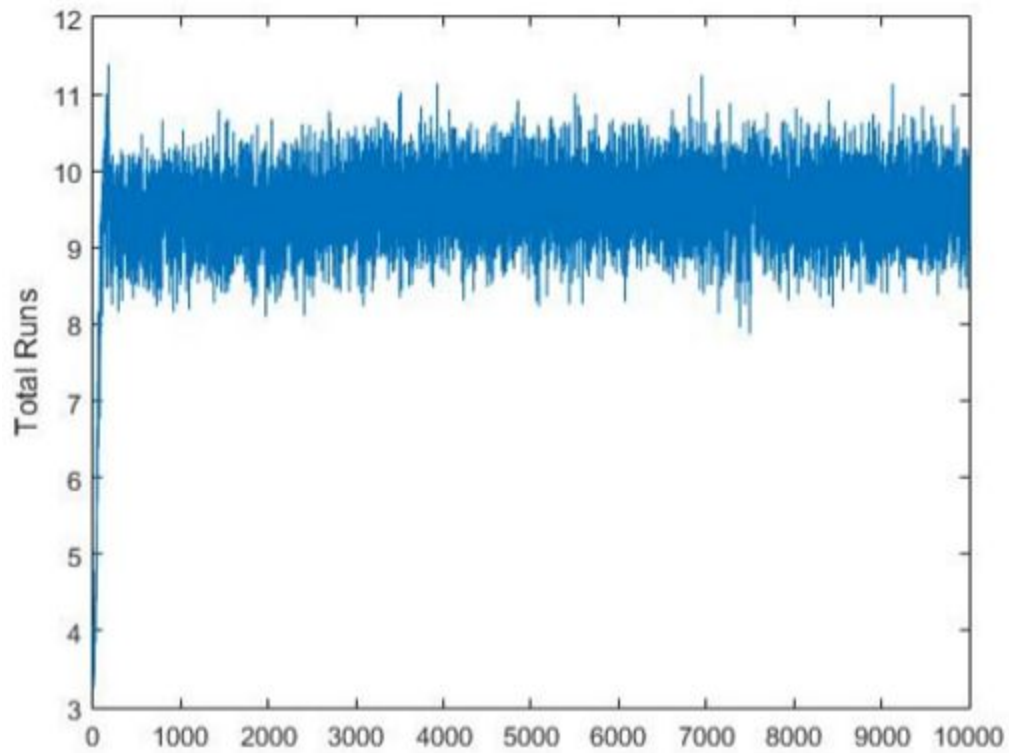


Results 1 Average Consecutive Hits Per Episode (Figure 10)



Average Consecutive Hits Per Episode (Figure 10)

The image above shows the result for a simulation instance with the total number of 10000 runs. It is evident from the figure that the number of hits is increased and then reaches to a suboptimal solution. As the number of runs is increased and more exploration is realized, we can observe that the agent's behaviour has converged to an optimal policy.

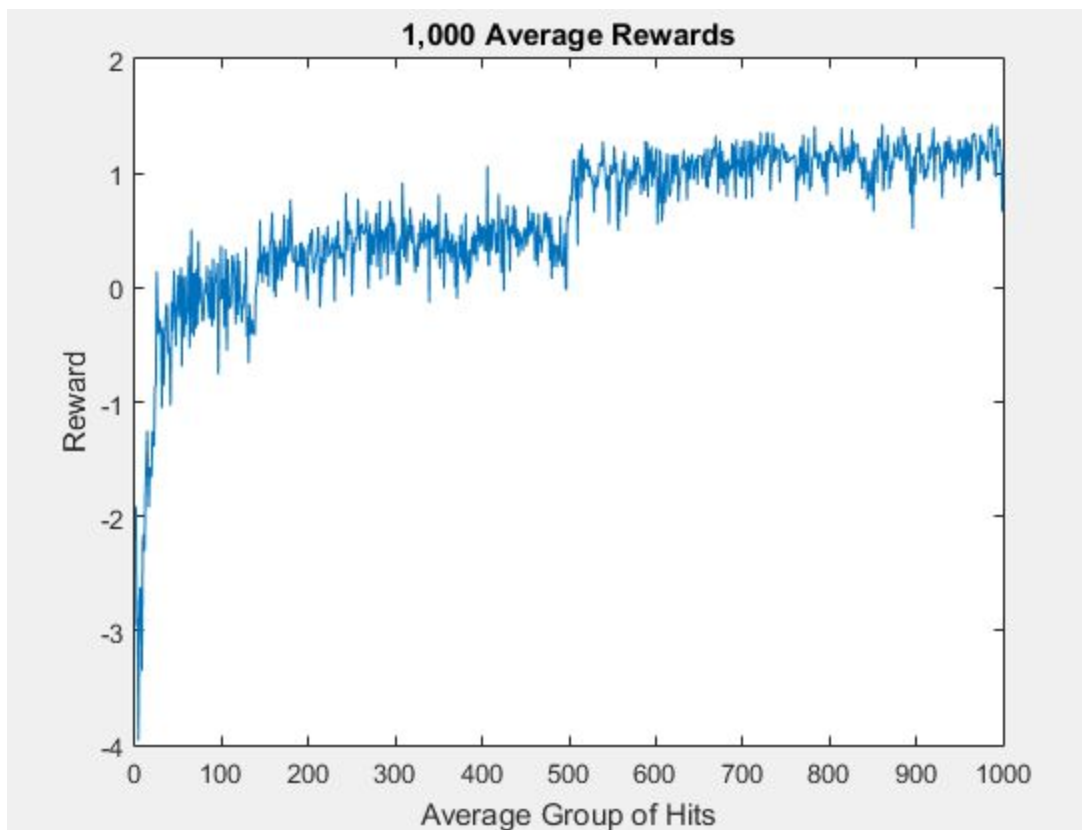


Average Consecutive Hits Per Episode for 10,000 hits (Figure 11)

In the figure above, it is evident that the agent learns in a rather fast amount of time and reaches a max threshold for the number of hits per run and the number of hits fluctuates about a fixed mean of about 9.6. It is noteworthy that each point of this graph is an average of 10000 successive runs.

4) Plotting average reward during the run time.

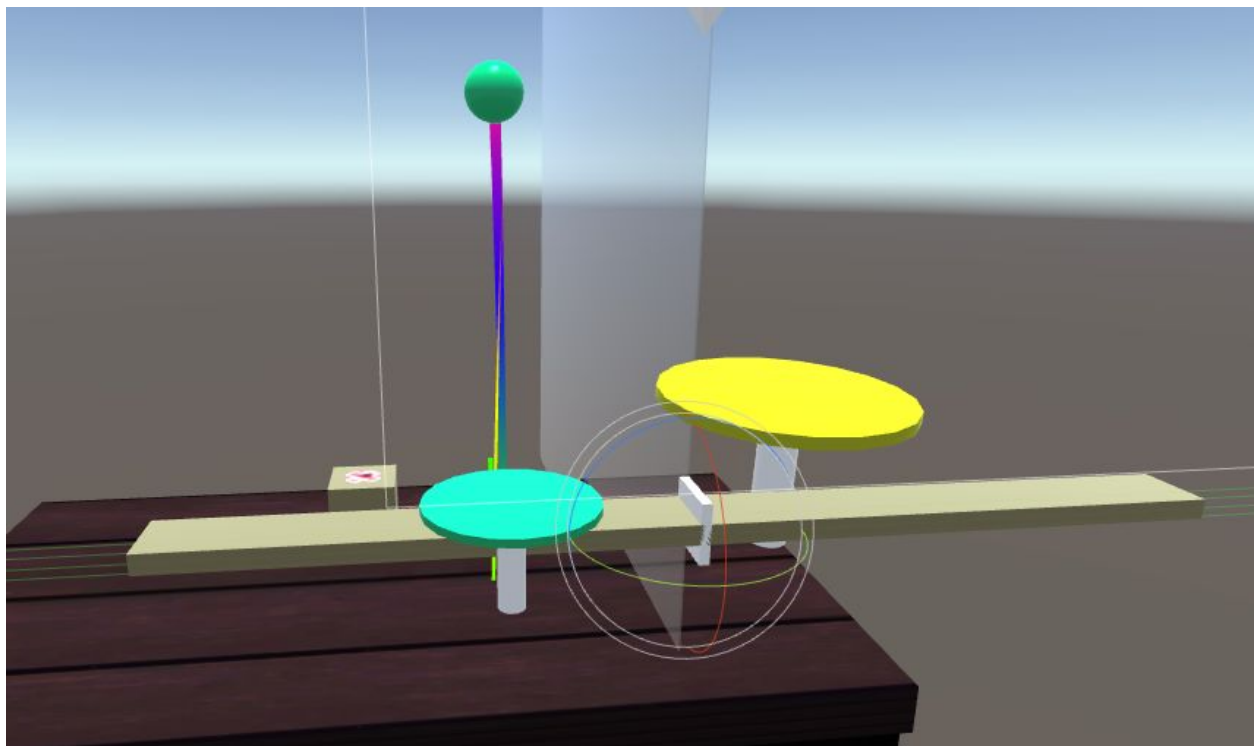
The final results from our learning algorithm are from interpreting the rewards. Number of hits can show how overall successful the agent is, but rewards can give more insight into if the agent is following the desired goals set for it.



Average rewards per 10 episodes for 10,000 episodes (Figure 12)

Above shows the agent taking actions that improve its reward over 10,000 episodes. -10 rewards are given to ball drops, +1 for successful hits, and +1 for hits that position the ball in the center on a continuous scale.

Unity Simulation:



Unity AI Bouncing Ball (Figure 13)

Above is a screenshot of the Unity agent bouncing the ball on its own. The AI is the small blue paddle and the human player is the large yellow one.

The AI performs the following actions:

1. When the ball intercepts the AI's 1 meter zone (the brown plank) the paddle moves to the ball's location

2. Once at the ball, the AI discretizes the continuous state space with the same algorithm used in our Matlab simulation
3. It gets the optimal action from matlabToUnity.m output and rotates to the correct angle and applies the correct transform with x and y velocities to the ball.

This implementation locks the agent and ball to the x axis. The agent is able to bounce the ball for around 5 hits depending on how the player knocks it into its zone. Its current iteration is bare bones and lacks important features like visualizing the impact velocities of the paddle and interpolating between paddle locations.

The agent does not bounce the ball “optimally”. While it is better at performing than a random action space, there is much room for improvement. Our Matlab results were able to improve performance but the could not fully account for the continuous simulation. With more time, we would attempt more further refine the number of states, actions, and learning method to further improve our results so they reach a policy more “optimal”.

Summary:

In this project a RL based solution for the VR ball bouncing was proposed and tested in MATLAB environment. The results indicate that the agent can learn how to play the game and hit the ball multiple times in a row. Finally, the resulting optimal actions for the state space were implemented into Unity and performed by a virtual agent.

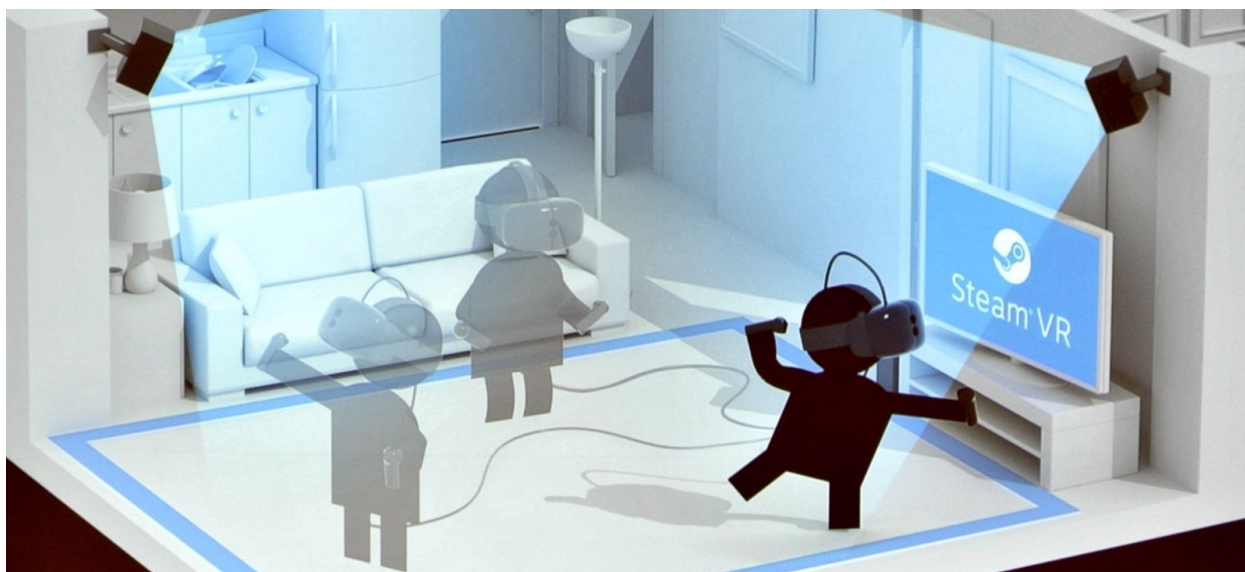
In order to achieve this, many tasks had to be completed. The state and action space had to be refined in order to for the project to be computationally feasible. A continuous physics simulation was made for collisions in both Matlab and Unity. The Unity simulation was hooked up to external tracking for player input recording. Multiple iterations of code refinement were performed in order to better improve the performance of the agent in the environment. For example changing the discrete action space was performed in order to ensure that there is no internal constraint for the agent hindering visitation to all states in the environment. A number of graphical means were used to visualize the results from the Q matrix. For example the number of successive hits were plotted in order to examine to which extent the algorithm is able to make the the ball hitting working.

Overall this project gave us the opportunity to increase our knowledge of machine learning specially Q-Learning and helped us gain hands on experience for implementation of learning algorithms. Our current results improve through learning but our solution is not as optimal as we imagined it could be. More iterations need to be done on our multiple design decisions in order to gauge the full potential of our model. Ultimately, we were able to implement the broad groundwork needed for an agent to learn to bounce a ball in virtual reality.

Future Work:**1. In-Engine Q-Learning:**

Instead of having the VR simulation and Matlab being separate, the Matlab code could be ported into C# to run inside of Unity. This has one important caveat: The learning simulation needs to run parallel to the VR simulation in order to not drop the FPS of the virtual user. Depending on implementation, this could limit the potential run speed of the learning agent but could allow for learning in parallel.

2. Room Scale Model Workshop:



Once the RL code is linked into the virtual environment it can then be modified in creative ways. With a room scale set up, different parts of the space could be devoted to interacting with the agent hyperparameters or model. For example, one area of the room could be devoted to painting reward functions while another could have levers for tweaking episode lengths. Interactive 3D learning plots could be displayed in addition to getting to interact with the resulting agent first hand after each iteration. While this may sound extravagant, there are potential benefits from manipulating data directly with your hands in order to develop an intuition for your system. Being able to stay in once space and constantly tinker with the model could be less taxing than constantly switching between windows to find the latest code and then waiting for long periods of time. If software based around a RL agent is released to the public, a workspace like this could allow easy modification and explanation of the internal functions and

lead to creative new solutions. Our current VR setup has very limited space and input so this design was not attempted. However, with all of the constant tweaking and evaluating we had to do, a streamlined obvious user interface would of been greatly appreciated, especially for collaboration.

Appendix A (Source code listing):

In this section, all MATLAB codes used for this project are provided.

simulation.m:

function [reward,nextxballhit,ballvxnext,ballvynext] = simulation(xballhit,ballvx,ballvy,i,j,k) % ,padangel,padvx,padvy these are actions that should not be in the code

```

rightlimit=10;
leftlimit=0;
global ballvylimit
ballvylimit=4;

g=9.81;

ballnormal=0.8;
padnormal=0.8;
padtangent=0.1;
balltangent=0.8;

ballangle=atan2(ballvy,ballvx);

%padangel=[-20,-10,0,10,20]; in degrees

if xballhit>=(leftlimit+rightlimit)/2
    Padangel=[0,10,20,30,40]*pi/180;
    %i
else
    Padangel=-[0,10,20,30,40]*pi/180; %i
end

if xballhit >=(leftlimit+rightlimit)/2;
    Padvx=[-2,-1,0,1,2]; %j
else
    Padvx=[2,1,0,-1,-2];

end

    Padvy=[-2,-1,0,1,2]; %k

if xballhit>(leftlimit+rightlimit)/2
    R=[cos(Padangel(i)) sin(Padangel(i)); -sin(Padangel(i)) cos(Padangel(i)) ];
    padvx=Padvx(j);
    padvy=Padvy(k);

```

```

gooo = R*[padvx;padvy];
padvt=gooo(1); padvn=gooo(2);

fooo= R*[ballvx;ballvy];
ballvt=fooo(1); ballvn=fooo(2);
ballvtAFTER=balltangent*ballvt+padtangent*padvt;
ballvnAFTER=-(ballvn*ballnormal)+padvn*padnormal;

Rinverse=[cos(Padangel(i)) -sin(Padangel(i)); sin(Padangel(i)) cos(Padangel(i)) ];
hooo=Rinverse*[ballvtAFTER;ballvnAFTER];

ballvxAFTER=hooo(1);
ballvyAFTER=hooo(2);% this is the velocity of ball right after impact

time_on_air=2*ballvyAFTER/g;

nextxballhit=xballhit+ ballvxAFTER*time_on_air;

ballvxnext=ballvxAFTER;
ballvynext=-ballvyAFTER;
end
if xballhit<=(leftlimit+rightlimit)/2

R=[cos(Padangel(i)) sin(Padangel(i)); -sin(Padangel(i)) cos(Padangel(i)) ];
padvx=Padvx(j);
padvy=Padvy(k);

gooo = R*[padvx;padvy];
padvt=gooo(1); padvn=gooo(2);

fooo= R*[ballvx;ballvy];
ballvt=fooo(1); ballvn=fooo(2);
ballvtAFTER=balltangent*ballvt+padtangent*padvt;
ballvnAFTER=-(ballvn*ballnormal)+padvn*padnormal;

Rinverse=[cos(Padangel(i)) -sin(Padangel(i)); sin(Padangel(i)) cos(Padangel(i)) ];
hooo=Rinverse*[ballvtAFTER;ballvnAFTER];

ballvxAFTER=hooo(1);
ballvyAFTER=hooo(2);% this is the velocity of ball right after impact

time_on_air=2*ballvyAFTER/g;

nextxballhit=xballhit+ ballvxAFTER*time_on_air;
ballvxnext=ballvxAFTER;
ballvynext=-ballvyAFTER;
end

if time_on_air<=0.1
    nextxballhit = rightlimit + 1;
    %reward=-1;
end

if (nextxballhit>rightlimit) || (nextxballhit<leftlimit) || (ballvylimit<ballvynext)

    reward=-1;
else
    reward=1;
end

end

```

disc.m :

This is simply a function which returns the discretized values of the results obtained from the simulation.m file.

```
function [Sxballhit, Sballvx, Sballvy]=disc(xballhit,ballvx,ballvy)
```

```
rightlimit=10;
leftlimit=0;
```

```
Sballvy = 7;
Sxballhit =7;
```

```
    if 0.5>xballhit && xballhit>=0
        fake=1;
        Sxballhit=5;

    elseif 1.5>xballhit && xballhit >=0.5
        fake=1;
        Sxballhit=4;

    elseif 2.5>xballhit && xballhit >=1.5
        fake=1;
        Sxballhit=3;

    elseif 3.5>xballhit && xballhit >=2.5
        fake=1;
        Sxballhit=2;

    elseif 4.5>xballhit && xballhit>=3.5
        fake=1;
        Sxballhit=1;

    elseif 5.5>xballhit && xballhit>=4.5
        fake=1;
        Sxballhit=6;

    elseif 6.5>xballhit && xballhit>=5.5
        fake=1;
        Sxballhit=1;

    elseif 7.5>xballhit && xballhit >=6.5
        fake=1;
        Sxballhit=2;

    elseif 8.5>xballhit && xballhit>=7.5
        fake=1;
        Sxballhit=3;

    elseif 9.5>xballhit && xballhit>=8.5
        fake=1;
        Sxballhit=4;

    elseif 10>xballhit && xballhit>=9.5
        fake=1;
        Sxballhit=5;
```



```

    end

%discreticize ballvx

if xballhit>=(leftlimit+rightlimit)/2;

if ballvx<-4.5
    Sballvx=11;

    elseif -4.5 <= ballvx && ballvx<-3.5

    Sballvx=10;

    elseif -3.5 <= ballvx && ballvx<-2.5
    Sballvx=9;

    elseif -2.5 <=ballvx && ballvx<-1.5
    Sballvx=8;

    elseif -1.5 <= ballvx && ballvx<-.5
    Sballvx=7;

    elseif -.5<=ballvx && ballvx<.5
    Sballvx=6;

    elseif .5<=ballvx && ballvx<1.5
    Sballvx=5;

    elseif 1.5<=ballvx && ballvx<2.5
    Sballvx=4;

    elseif 2.5<=ballvx && ballvx<3.5
    Sballvx=3;

    elseif 3.5<=ballvx && ballvx<4.5
    Sballvx=2;

    elseif 4.5<ballvx || ballvx==4.5;
    Sballvx=1;

end

end

if xballhit<(leftlimit+rightlimit)/2;

if ballvx<-4.5
    Sballvx=1;

    elseif -4.5 <= ballvx && ballvx<-3.5
    Sballvx=2;
    elseif -3.5 <= ballvx && ballvx<-2.5

```

```

Sballvx=3;

elseif -2.5 <=ballvx && ballvx<-1.5
Sballvx=4;

elseif -1.5 <= ballvx && ballvx<-.5
Sballvx=5;

elseif -.5<=ballvx && ballvx<.5
Sballvx=6;

elseif .5<=ballvx && ballvx<1.5
Sballvx=7;

elseif 1.5<=ballvx && ballvx<2.5
Sballvx=8;

elseif 2.5<=ballvx && ballvx<3.5
Sballvx=9;

elseif 3.5<=ballvx && ballvx<4.5
Sballvx=10;

elseif 4.5<ballvx || ballvx==4.5;
Sballvx=11;

end
end

% discretize ballvy

% if ballvy> 0 || ballvy==0
% disp('error')

if -.5<ballvy && ballvy<0 || ballvy==-.5;
Sballvy=6;

elseif -1.5<ballvy && ballvy<-.5 || ballvy==-.5;
Sballvy=5;

elseif -2.5<ballvy && ballvy<-1.5 || ballvy==-.5;
Sballvy=4;

elseif -3.5<ballvy && ballvy<-2.5 || ballvy==-.5;
Sballvy=3;

elseif -4.5<ballvy && ballvy<-3.5 || ballvy==-.5;
Sballvy=2;

elseif ballvy<-4.5
Sballvy=1;

end,end

```

The code for Q- Learning is provided in below.

qlearning.m

```
%clear all
clc
tic
runs=100000;

action1num=5;
action2num=5;
action3num=5;

state1num=7;
state2num=11;
state3num=7;

gamma=0.8;
alpha=0.5;

lastaction1=0;
lastaction2=0;
lastaction3=0;
lstate1=0;
lstate2=0;
lstate3=0;

global ballvylimit
ballvylimit=-.5;

Aruns = zeros(10000);
firstTenR = zeros(10000);
runNumb = 0;
intI = 1;
runSums = 10;

totalRunsCnt = 0;
xballhit=20;
Q=zeros(action1num,action2num,action3num,state1num,state2num,state3num);

previousxHit = -1;

for run=1:runs;

    if mod(run, runSums) == 0
        Aruns(intI) = totalRunsCnt/runSums;
        totalRunsCnt = 0;
        intI = intI + 1;
    end

    %%%%%%%%%%give initial values to states and actions
    %xballhit=rand(1)*10;
    xballhit=3 + rand(1)*6;
    ballvx1=rand(1)*3;
    ballvx2=rand(1)*-3;
    %ballvx1=rand(1)*7;
    %ballvx2=rand(1)*-7;
    if abs(ballvx1)>abs(ballvx2)
        ballvx=ballvx1;
    else
```

```

        ballvx=ballvx2;
    end
    ballvx;
    %ballvy=rand(1)*-5;
    ballvy=-1 + rand(1)*-2;
    % get the states for the initial conditions

    stepnum=10; % how many steps in the episode

    reward=0;
    % find the greedy action or choose randomly
    for step=1:stepnum

        [Sxballhit0, Sballvx0, Sballvy0]=disc(xballhit,ballvx,ballvy);

        if xballhit>=0 && xballhit<=10 && ballvy<ballvylimit
            totalRunsCnt = totalRunsCnt+1;
            actionsForGivenState = Q(:, :, :, Sxballhit0, Sballvx0, Sballvy0);
            %1 dementionalize and find max
            M = actionsForGivenState;%= randn(10,10,10,10);
            [C,I] = max(M(:));
            temp = find(max(M(:)) == M(:));

            [C,I] = max(M(:));
            C = C;
            randI = temp(randi([1 length(temp)]));
            M(randI);

            [I1,I2,I3] = ind2sub(size(M),randI);
            M(I1,I2,I3);
            maxAction1 = I1;
            maxAction2 = I2;
            maxAction3 = I3;
            optimalActionValue = C;

        [reward,nextxballhit,ballvxnext,ballvynext] = simulation(xballhit,ballvx,ballvy,I1,I2,I3);
        %
        % if(previousxHit == -1)
        % PlotHit( xballhit, xballhit, nextxballhit, I1, I2, I3,ballvxnext ,0)
        % else
        % PlotHit( previousxHit, xballhit, nextxballhit, I1, I2, I3,ballvxnext ,ballvx)
        % end

        if(runNumb<10000)
            runNumb = runNumb + 1;
            firstTenR(runNumb) = reward;
        end

        lastaction1=I1;
        lastaction2=I2;
        lastaction3=I3;

        %nextxballhit,ballvxnext,ballvynext
        %PlotHit( previousPadX, currentPadX, nextPadX, PadAngle, PaddleXVelocity, PaddleYVelocity,vyoutput
        ,vyinput)

        [Sxballhit, Sballvx, Sballvy]=disc(nextxballhit,ballvxnext,ballvynext);
        lstate1=Sxballhit0;
        lstate2=Sballvx0;

```

```

lstate3=Sballvy0;

    actionsForGivenState = Q(:, :, :, Sxballhit, Sballvx, Sballvy);
    %1 dementionalize and find max
M = actionsForGivenState; % = randn(10,10,10,10);
[C,I] = max(M(:));
temp = find(max(M(:)) == M(:));

[C,I] = max(M(:));
C = C;
randI = temp(randi([1 length(temp)]));
M(randI);

[I1,I2,I3] = ind2sub(size(M),randI);
M(I1,I2,I3);
NextmaxAction1 = I1;
NextmaxAction2 = I2;
NextmaxAction3 = I3;
optimalActionValue = C;

delta=reward+gamma*Q(NextmaxAction1,NextmaxAction2,NextmaxAction3,Sxballhit, Sballvx,
Sballvy)-Q(maxAction1,maxAction2,maxAction3,Sxballhit0, Sballvx0, Sballvy0);
%E(row,col,optimalA) = E(row,col,optimalA) + 1;
%E(row,col,optimalA) = (1-alpha)*E(row,col,optimalA) + 1
Q(I1,I2,I3,Sxballhit0, Sballvx0, Sballvy0)=Q(I1,I2,I3,Sxballhit0, Sballvx0, Sballvy0)+alpha*delta;

AAA=[nextxballhit,ballvxnext,ballvynext];
previousxHit = xballhit;
xballhit=AAA(1);
ballvx=AAA(2);
ballvy=AAA(3);

end

end

end
figure
x=(1:10000);
plot(x,Aruns(1:10000))
xlabel('500 Runs')
ylabel('Total Runs')
title('50,000 15 Hit Runs - Smart PVs')

figure
x=(1:10000);
plot(x,firstTenR(1:10000))
xlabel('Hit')
ylabel('Reward')
title('First 10,000 Rewards')

toc
%disp(Q(lastaction1,lastaction2,lastaction3,lstate1,lstate2,lstate3));

```

The code for exporting the optimal policy. It was manually converted in Excel to an array of strings seen in physics.cs

matlabToUnity.cs

The code

```

fileID = fopen('optimalAct.csv','w');
state1num=7;
state2num=11;
state3num=7;

for state1=1:state1num-1%doesn't print failure states
    for state2=1:state2num
        for state3=1:state3num-1

            actionsForGivenState = Q(:, :, :, state1, state2, state3);
            %1 dementionalize and find max
            M = actionsForGivenState;%= randn(10,10,10,10);
            [C,I] = max(M(:));
            temp = find(max(M(:)) == M(:));

            [C,I] = max(M(:));
            %C = C;
            randI = temp(randi([1 length(temp)]));
            M(randI);

            [I1,I2,I3] = ind2sub(size(M),randI);
            M(I1,I2,I3);
            maxAction1 = I1;
            maxAction2 = I2;
            maxAction3 = I3;

            if(Q(maxAction1,maxAction2,maxAction3,state1, state2, state3) <= 0)
                %fprintf(fileID, '%i,%i,%i V:%d\n', maxAction1,maxAction2,maxAction3,Q(maxAction1,maxAction2,maxAction3,state1, state2, state3),state1,state2,state3);
                fprintf(fileID, '%i,%i,%i,%d,%i,%i,%i\n', -1, -1, -1, Q(maxAction1,maxAction2,maxAction3,state1, state2, state3), state1, state2, state3);
            else

                fprintf(fileID, '%i,%i,%i,%d,%i,%i,%i\n', maxAction1,maxAction2,maxAction3,Q(maxAction1,maxAction2,maxAction3,state1, state2, state3),state1,state2,state3);

            end

        end
    end
end
end

```

The code for Unity physics + AI physics.cs

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System;

public class physics : MonoBehaviour {
    public float thrust;
    public Rigidbody rb;
    public Rigidbody PaddleRb;
    public GameObject tcpNetworking;
    public GameObject myPaddle;
    public GameObject bubble;
    public GameObject my2DWall;
    public GameObject diamond;

    public GameObject AIMiddle;
    public GameObject AIPaddle;
    public GameObject AITop;

    Vector3 startpos;
    public float ylimit = -11;
    public float startYvel;

    Vector3 ballV;
    Vector3 relBallV;

    float magnitude;
    public float fixedMagnitude;
    public bool useFixedMag;

    public float angle = 0.0F;
    public AudioSource[] audio;
    private Vector3 relative;
    private Vector3 flipped;
    public Vector3 startVelocity;
    public float myGravity;

    Transform oldTransform;
    Transform newTransform;
    Vector3 paddleVel;

    TrailRenderer myTrail;
    public float trailTime;

    tcp tcpScript;
    bubbleCollider bubbleScript;
    getRoll myRoll;

    int score;
    int lastScore;
    int highScore;
    int level;
    int countdown;
    public int countDownSpeed;
    public int hitsPerLevel;
    public float minHitTime;
    float hitTime;
    int totalHits;

```

```

bool restart;

public Material paddleMat;
public Material ballMat;
public Camera myCamera;
Color[] colorArray = new Color[12];
float colorLerp;
public float colorLerpValue;

public TextMesh levelDisplay;
public TextMesh scoreDisplay;
public TextMesh lastScoreDisplay;
public TextMesh highScoreDisplay;
public TextMesh ballDisplay;

int ballsDropped;
public int levelDownBallDrops;
int totalBallsDropped;
public int totalBallLives;

public float startheight = .65f;
public float myTimeScale = 1f;

float roll;

// Start is called at first frame
void Start()
{
    initializeSave();
    Time.timeScale = myTimeScale;
    level = 1;
    colorLerp = 1;
    #region COLOR
    colorArray[0] = new Color(1, 0, 0);
    colorArray[1] = new Color(1, .5f, 1);
    colorArray[2] = new Color(1, 1, 0);
    colorArray[3] = new Color(.5f, 1, 0);
    colorArray[4] = new Color(0, 1, 0);
    colorArray[5] = new Color(0, 1, .5f);
    colorArray[6] = new Color(0, 1, 1);
    colorArray[7] = new Color(0, .5f, 1);
    colorArray[8] = new Color(0, 0, 1);
    colorArray[9] = new Color(.5f, 0, 1);
    colorArray[10] = new Color(1, 0, 1);
    colorArray[11] = new Color(1, 0, .5f);
    #endregion

    restart = true;
    rb = GetComponent<Rigidbody>();
    PaddleRb = myPaddle.GetComponent<Rigidbody>();
    audio = GetComponent<AudioSource>();
    myTrail = GetComponent<TrailRenderer>();
    tcpScript = tcpNetworking.GetComponent<tcp>();
    bubbleScript = bubble.GetComponent<bubbleCollider>();
    myRoll = myPaddle.GetComponent<getRoll>();
    //myCamera = GetComponent<Camera>();

    trailTime = myTrail.time;
    startpos = transform.position;
    //rb.velocity = startVelocity;
    newTransform = myPaddle.transform;
    totalHits = 0;
    hitTime = Time.time;
    ballsDropped = 0;

```



```

    LevelChange(0);
    updateBalls(0);
    highScore = 0;
}

// Update is called once per frame
void Update()
{

    Vector3 pos = ProjectPointOnPlane(Vector3.up, Vector3.zero, transform.forward);
    roll = SignedAngle(myPaddle.transform.right, pos, myPaddle.transform.forward);

    Time.timeScale = myTimeScale;
    //Temporary color lerp
    if (colorLerp <= 1)
    {
        paddleMat.color = Color.Lerp(colorArray[(level - 1 + 1) % 12], colorArray[(level - 1 + 1 + 1)
% 12], colorLerp);
        ballMat.color = Color.Lerp(colorArray[(level - 1 + 4) % 12], colorArray[(level - 1 + 4 + 1) %
12], colorLerp);
        myCamera.backgroundColor = Color.Lerp(colorArray[(level - 1 + 8) % 12], colorArray[(level - 1
+ 8 + 1) % 12], colorLerp);
        colorLerp = colorLerp + colorLerpValue;
    }
    //Ball reset
    if (transform.position.y < ylimit)
        ballreset();
    if (Input.GetMouseButtonDown(2))
    {
        Debug.Log("saving output");
        outputSave();
        //ballreset();
    }
    if (Input.GetButtonDown("Set"))
        setStartPos(myPaddle.transform.position + Vector3.up * .2f);
    if (Input.GetMouseButtonDown(1))
        setStartPos(myPaddle.transform.position + Vector3.up * .2f);

}

Vector3 ProjectPointOnPlane(Vector3 planeNormal, Vector3 planePoint, Vector3 point)
{
    planeNormal.Normalize();
    float distance = -Vector3.Dot(planeNormal.normalized, (point - planePoint));
    return point + planeNormal * distance;
}

float SignedAngle(Vector3 v1, Vector3 v2, Vector3 normal)
{
    Vector3 perp = Vector3.Cross(normal, v1);
    float angle = Vector3.Angle(v1, v2);
    angle *= Mathf.Sign(Vector3.Dot(perp, v2));
    return angle;
}

void FixedUpdate()
{
    //Gravity
    if (!restart)
        rb.AddForce(transform.up * myGravity);
}

void LevelChange(int addLevel)

```

```

{
    if (level + addLevel < 1)
        level = 1;
    else
    {
        if (level < level + addLevel)
        {
            audio[1].Play();
        }
        else if (level > level + addLevel)
        {
            audio[2].Play();
        }

        level = level + addLevel;
    }
    myGravity = -9.81f; //myGravity = -(1 + level / 2);
    //startVelocity = Vector3.up * -(myGravity / 1.42f);
    startVelocity = Vector3.up * Mathf.Sqrt(Mathf.Abs(2 * myGravity * startheight));
    levelDisplay.text = "LEVEL: " + level.ToString();
    colorLerp = 0;
    ballsDropped = 0;
}

```

```

void LateUpdate()
{
    oldTransform = newTransform;
    newTransform = myPaddle.transform;
    //Debug.Log("old      : " + oldTransform.position);
    //Debug.Log("new      : " + newTransform.position);
    paddleVel = (newTransform.position - oldTransform.position) / Time.deltaTime;
    //Debug.Log("PaddleVel : " + paddleVel);
    //Debug.Log("RB PaddleVel: " + PaddleRb.velocity);
}

```

```

Vector3 myPaddleVel;
Vector3 myPaddleAng;
public float YballElasticity = 1;
public float XZballElasticity = 1;
public float YpaddleTransfer = 1;
public float XZpaddleTransfer = 1;

void OnCollisionEnter(Collision collision)
{
    Debug.Log("Collision entered");
}

```

```

Vector3 relativePaddleVel;
void OnTriggerEnter(Collider collision)
{
    if (collision.CompareTag("Diamond"))
    {
        diamondHit();
    }
    if (collision.CompareTag("Paddle"))
    {
        // Debug.Log("Time between hits: " + (Time.time - hitTime).ToString());
        if (Time.time - hitTime > minHitTime)
        {
            totalHits = totalHits + 1;
            updateScore(10 * (1 + level / 3));
        }
    }
}

```

```

}

if (totalHits >= (hitsPerLevel + level))
{
    totalHits = 0;
    LevelChange(1);
}
hitTime = Time.time;

// Debug.Log("Trigger entered");
// Debug.Log("PaddleVel : " + paddleVel);
myPaddleVel = tcpScript.paddleVel;
// Debug.Log("TCP vel : " + myPaddleVel);
//Debug.Break();
ballV = rb.velocity;
if (useFixedMag)
    magnitude = fixedMagnitude;
else
    magnitude = ballV.magnitude;

//Flip the velocity relative to the paddle
relative = myPaddle.transform.InverseTransformDirection(ballV);
relativePaddleVel = myPaddle.transform.InverseTransformDirection(myPaddleVel);

//Debug.Log("Rel P vel : " + relativePaddleVel);
flipped = new Vector3((relative.x * XZballElasticity) + (relativePaddleVel.x *
XZpaddleTransfer), (-relative.y * YballElasticity) + (relativePaddleVel.y * YpaddleTransfer), (relative.z
* XZballElasticity) + (relativePaddleVel.x * XZpaddleTransfer));
#region DEBUG
/*
    Debug.Log("relative : " + relative);
    Debug.Log("unflipped : " + myPaddle.transform.TransformDirection(flipped));
*/
#endregion
audio[0].volume = (.2f + (flipped.magnitude / rb.velocity.magnitude)) * .2f;
//Debug.Log("change in magnitude ratio " + flipped.magnitude / rb.velocity.magnitude);

string[] collisionInfo = new string[14];
myPaddleAng = tcpScript.paddleAng;
//Debug.Log("Xang: " + myPaddleAng.x * Mathf.Sign(myPaddle.transform.up.x) + " Xang2: " +
Mathf.Atan(myPaddle.transform.up.y / myPaddle.transform.up.x) * Mathf.Rad2Deg);
collisionInfo[0] = (startpos.x - transform.position.x).ToString(); // "Xball";
collisionInfo[1] = (startpos.z - transform.position.z).ToString(); // "ZBall";
collisionInfo[2] = (rb.velocity.x).ToString(); // "BallVX";
collisionInfo[3] = (rb.velocity.z).ToString(); // "BallVZ";
collisionInfo[4] = (rb.velocity.y).ToString(); // "BallVY";
collisionInfo[5] = (myPaddleAng.x * Mathf.Sign(myPaddle.transform.up.x) - 90).ToString(); //
"PaddleAngX";
collisionInfo[6] = (myPaddleAng.x * Mathf.Sign(myPaddle.transform.up.z) - 90).ToString(); //
"PaddleAngZ";
collisionInfo[7] = (myPaddleVel.x).ToString(); // "PaddleVX";
collisionInfo[8] = (myPaddleVel.z).ToString(); // "PaddleVZ";
collisionInfo[9] = (myPaddleVel.y).ToString(); // "PaddleVY";
collisionInfo[10] = (Mathf.Atan(myPaddle.transform.up.y / myPaddle.transform.up.x) *
Mathf.Rad2Deg).ToString(); // "PaddleAngXv2";
collisionInfo[11] = (myPaddle.transform.TransformDirection(flipped).x).ToString(); //
"BallVXOut";
collisionInfo[12] = (myPaddle.transform.TransformDirection(flipped).z).ToString(); //
"BallVZOut";
collisionInfo[13] = (myPaddle.transform.TransformDirection(flipped).y).ToString(); //
"BallVYOut";
saveData(collisionInfo);

Debug.Log("PADDLE");
Debug.Log("Ball X Pos " + (transform.position.x - AIMiddle.transform.position.x));

```

```

        Debug.Log("Ball X Vel " + rb.velocity.x);
        Debug.Log("Ball Y Vel " + rb.velocity.y);

        Debug.Log("Ball X Pos Discrete #" + getDiscretePosition((transform.position.x -
AIMiddle.transform.position.x)));
        Debug.Log("Ball X Vel Discrete #" + getDiscreteXVelocity(rb.velocity.x, (transform.position.x
- AIMiddle.transform.position.x)));
        Debug.Log("Ball Y Vel Discrete #" + getDiscreteYVelocity(rb.velocity.y));

        rb.velocity = myPaddle.transform.TransformDirection(flipped);
        //rb.AddForce(bounceForce * thrust);
        //Debug.Log("audio volume" + audio[0].volume);
        // Debug.Log("hit velocity magnitude" + rb.velocity.magnitude);
        //Debug.Log("hit velocity magnitude / g" + .5f*(rb.velocity.magnitude*rb.velocity.magnitude)
/myGravity);

        audio[0].Play();

        //Get distance to center

        //Get change in angle

        //Apply angular momentum

        //Apply
    }

    if (collision.CompareTag("AI"))
    {
        Debug.Log("--AI--");
        Debug.Log("Ball X Pos " + (transform.position.x - AIMiddle.transform.position.x));
        Debug.Log("Ball X Vel " + rb.velocity.x);
        Debug.Log("Ball Y Vel " + rb.velocity.y);
        int xpos = getDiscretePosition((transform.position.x - AIMiddle.transform.position.x));
        int xvel = getDiscreteXVelocity(rb.velocity.x, (transform.position.x -
AIMiddle.transform.position.x));
        int yvel = getDiscreteYVelocity(rb.velocity.y);

        Debug.Log("Ball X Pos Discrete #" + getDiscretePosition((transform.position.x -
AIMiddle.transform.position.x)));
        Debug.Log("Ball X Vel Discrete #" + getDiscreteXVelocity(rb.velocity.x, (transform.position.x
- AIMiddle.transform.position.x)));
        Debug.Log("Ball Y Vel Discrete #" + getDiscreteYVelocity(rb.velocity.y));

        String actions = getAction(xpos, xvel, yvel);
        Debug.Log("Optimal Actions" + actions);

        float[] paddleAngles = new float[5] {0, 3, 7, 10, 30};
        float[] paddleXVelocities = new float[5] { -.3f, -.1f, 0, .1f, .3f };
        float[] paddleYVelocities = new float[5] { 0, .1f, .15f, .3f, 1 };

        //float paddleAng = paddleAngles[actions[0]].

        Debug.Log(actions.Length);

        Debug.Log("Action to take " + paddleAngles[actions[1] - 1 - '0'] + " " +
paddleXVelocities[actions[2] - 1 - '0'] + " " + paddleYVelocities[actions[3] - 1 - '0']);

```

```

        AIPaddle.transform.position = new Vector3(transform.position.x, AIPaddle.transform.position.y,
AIPaddle.transform.position.z);
        //bubble.transform.position = new Vector3(startpos.x, myPaddle.transform.position.y + .2f,
startpos.z)

        AITop.transform.localRotation = Quaternion.Euler(new Vector3(AITop.transform.rotation.x,
AITop.transform.rotation.y, paddleAngles[actions[1] - 1 - '0'] * Math.Sign((transform.position.x -
AIMiddle.transform.position.x))));

        float AIXvel = paddleXVelocities[actions[2] - 1 - '0'];
        float AIYvel = paddleYVelocities[actions[3] - 1 - '0'];
        ballV = rb.velocity;
        //Flip the velocity relative to the paddle
        relative = AITop.transform.InverseTransformDirection(ballV);
        relativePaddleVel = myPaddle.transform.InverseTransformDirection(myPaddleVel);

        //Debug.Log("Rel P vel      :" + relativePaddleVel);
        flipped = new Vector3((relative.x * XZballElasticity) + (AIXvel * XZpaddleTransfer),
(-relative.y * YballElasticity) + (AIYvel * YpaddleTransfer), (relative.z * XZballElasticity) + (AIXvel *
XZpaddleTransfer));

        rb.velocity = myPaddle.transform.TransformDirection(flipped);

    }

}

void updateScore(int addScore)
{
    score = score + addScore;
    scoreDisplay.text = "SCORE: " + score.ToString();
}

void updateBalls(int addBalls)
{
    totalBallsDropped = totalBallsDropped + addBalls;
    int ballsLeft = (totalBallLives - totalBallsDropped);
    ballDisplay.text = "BALLS: " + ballsLeft.ToString();
}

}

public void playPop()
{
    audio[3].Play();
}

//Sets the ball back to its starting pos after it falls
void ballreset()
{
    totalHits = 0;
    bubbleScript.createBubble();
    restart = true;
    ballsDropped = ballsDropped + 1;
    updateBalls(-1);
    if (ballsDropped > levelDownBallDrops)
    {
        LevelChange(-1);
    }

    if (totalBallsDropped >= totalBallLives)
    {
        gameOver();
    }
}

```

```

    }

    myTrail.time = 0;
    transform.position = new Vector3(startpos.x, myPaddle.transform.position.y + .2f, startpos.z);
    bubble.transform.position = new Vector3(startpos.x, myPaddle.transform.position.y + .2f,
startpos.z);

    rb.velocity = Vector3.zero;
    //
    myTrail.time = trailTime;
}

public void ballstart()
{
    rb.velocity = startVelocity;
    restart = false;
    setDiamondLoc();
}

void gameOver()
{
    audio[4].Play();
    lastScore = score;
    lastScoreDisplay.text = "LAST\nSCORE\n" + lastScore.ToString();
    score = 0;
    scoreDisplay.text = "SCORE: " + score.ToString();
    if (highScore < lastScore)
    {
        audio[5].Play();
        highScore = lastScore;
        highScoreDisplay.text = "HIGH\nSCORE\n" + highScore.ToString();
    }
    totalBallsDropped = 0;
    ballDisplay.text = "BALLS: " + totalBallLives.ToString();
    level = 1;
    myGravity = -9.81f; // myGravity = -(1 + level / 2);
    startVelocity = Vector3.up * Mathf.Sqrt(Mathf.Abs(2 * myGravity * startheight));
    levelDisplay.text = "LEVEL: " + level.ToString();
}

void setStartPos(Vector3 location)
{
    //Debug.Log("Start location set: " + location);
    bubble.transform.position = location;
    my2DWall.transform.position = location;
    transform.position = location;
    startpos = location;
}

void diamondHit()
{
    updateScore(15 * (1 + level));
    audio[6].Play();
    setDiamondLoc();
}

void setDiamondLoc()
{
    float randMag;
    float randTheta;

    randMag = UnityEngine.Random.Range(.05F, .13F);
    randTheta = UnityEngine.Random.Range(0, 360F);
    float randX = randMag * Mathf.Cos(randTheta);

```

```

        float randZ = randMag * Mathf.Sin(randTheta);
        float randY = UnityEngine.Random.Range(.13F, .4F);
        diamond.transform.position = new Vector3(startpos.x + randX, startpos.y + randY, startpos.z +
randZ);
    }

```

```

private List<string[]> rowData = new List<string[]>();
string[] rowDataTemp = new string[14];
void initializeSave()
{
    // Creating First row of titles manually..

    rowDataTemp[0] = "Xball";
    rowDataTemp[1] = "ZBall";
    rowDataTemp[2] = "BallVX";
    rowDataTemp[3] = "BallVZ";
    rowDataTemp[4] = "BallVY";
    rowDataTemp[5] = "PaddleAngX";
    rowDataTemp[6] = "PaddleAngZ";
    rowDataTemp[7] = "PaddleVX";
    rowDataTemp[8] = "PaddleVZ";
    rowDataTemp[9] = "PaddleVY";
    rowDataTemp[10] = "PaddleAngXv2";
    rowDataTemp[11] = "BallVXOut";
    rowDataTemp[12] = "BallVZOut";
    rowDataTemp[13] = "BallVYOut";
    rowData.Add(rowDataTemp);
}
void saveData(string[] input) {
    rowData.Add(input);
}

void outputSave()
{
    // You can add up the values in as many cells as you want.

    string[][] output = new string[rowData.Count][];

    for (int i = 0; i < output.Length; i++)
    {
        output[i] = rowData[i];
    }

    int length = output.GetLength(0);
    string delimiter = ",";

    StringBuilder sb = new StringBuilder();

    for (int index = 0; index < length; index++)
        sb.AppendLine(string.Join(delimiter, output[index]));

    string filePath = getPath();
    Debug.Log("Output File Path: " + filePath);

    StreamWriter outputStream = System.IO.File.CreateText(filePath);
    outputStream.WriteLine(sb);
    outputStream.Close();
}

// Following method is used to retrieve the relative path as device platform
private string getPath()
{

```

```

#ifdef UNITY_EDITOR
    return Application.dataPath + "/CSV/" + "Saved_data.csv";
#elif UNITY_ANDROID
    return Application.persistentDataPath+"Saved_data.csv";
#elif UNITY_IPHONE
    return Application.persistentDataPath+"/"+ "Saved_data.csv";
#else
    return Application.dataPath + "/"+"Saved_data.csv";
#endif
}
/*
float getRoll(Vector3 originalTransform)
{
    GameObject tempGO = new GameObject();
    Transform t = tempGO.transform;
    t.localRotation = Quaternion.Euler(new Vector3(originalTransform.x, originalTransform.y,
originalTransform.z));

    t.Rotate(0, 0, t.localEulerAngles.z * -1);

    GameObject.Destroy(tempGO);
    return t.localEulerAngles.x;
}
*/

int getDiscretePosition(float xballhit)
{
    xballhit = 5 + xballhit * 10;

    int Sxballhit = 6;
    if (0.5 > xballhit && xballhit >= 0)
        Sxballhit = 5;
    else if (1.5 > xballhit && xballhit >= 0.5)
        Sxballhit = 4;
    else if (2.5 > xballhit && xballhit >= 1.5)
        Sxballhit = 3;
    else if (3.5 > xballhit && xballhit >= 2.5)
        Sxballhit = 2;
    else if (4.5 > xballhit && xballhit >= 3.5)
        Sxballhit = 1;
    else if (5.5 > xballhit && xballhit >= 4.5)
        Sxballhit = 6;
    else if (6.5 > xballhit && xballhit >= 5.5)
        Sxballhit = 1;
    else if (7.5 > xballhit && xballhit >= 6.5)
        Sxballhit = 2;
    else if (8.5 > xballhit && xballhit >= 7.5)
        Sxballhit = 3;
    else if (9.5 > xballhit && xballhit >= 8.5)
        Sxballhit = 4;
    else if (10 > xballhit && xballhit >= 9.5)
        Sxballhit = 5;

    return Sxballhit;
}

int getDiscreteXVelocity(float ballvx, float xballhit)
{
    int Sballvx = 11;
    float leftlimit = 0;
    float rightlimit = 10;

    float ballvxTemp;

    if (xballhit >= (leftlimit + rightlimit) / 2)

```



```

{
    ballvxTemp = ballvx * 5;

    if (ballvxTemp < -4.5)
Sballvx = 11;

        else if (-4.5 <= ballvxTemp && ballvxTemp < -3.5)
Sballvx = 10;

            else if (-3.5 <= ballvxTemp && ballvxTemp < -2.5)
Sballvx = 9;

                else if (-2.5 <= ballvxTemp && ballvxTemp < -1.5)
Sballvx = 8;

                    else if (-1.5 <= ballvxTemp && ballvxTemp < -.5)
Sballvx = 7;

                        else if (-.5 <= ballvxTemp && ballvxTemp < .5)
Sballvx = 6;

                            else if (.5 <= ballvxTemp && ballvxTemp < 1.5)
Sballvx = 5;

                                else if (1.5 <= ballvxTemp && ballvxTemp < 2.5)
Sballvx = 4;

                                    else if (2.5 <= ballvxTemp && ballvxTemp < 3.5)
Sballvx = 3;

                                        else if (3.5 <= ballvxTemp && ballvxTemp < 4.5)
Sballvx = 2;

                                            else if (4.5 < ballvxTemp || ballvxTemp == 4.5)
Sballvx = 1;

}

if (xballhit < (leftlimit + rightlimit) / 2)
{
    ballvxTemp = ballvx * 10;

    if (ballvxTemp < -4.5)

        Sballvx = 1;

    else if (-4.5 <= ballvxTemp && ballvxTemp < -3.5)

        Sballvx = 2;

    else if (-3.5 <= ballvxTemp && ballvxTemp < -2.5)
        Sballvx = 3;

    else if (-2.5 <= ballvxTemp && ballvxTemp < -1.5)
        Sballvx = 4;

    else if (-1.5 <= ballvxTemp && ballvxTemp < -.5)
        Sballvx = 5;

    else if (-.5 <= ballvxTemp && ballvxTemp < .5)

```

```

        Sballvx = 6;

    else if (.5 <= ballvxTemp && ballvxTemp < 1.5)
        Sballvx = 7;

    else if (1.5 <= ballvxTemp && ballvxTemp < 2.5)
        Sballvx = 8;

    else if (2.5 <= ballvxTemp && ballvxTemp < 3.5)
        Sballvx = 9;

    else if (3.5 <= ballvxTemp && ballvxTemp < 4.5)
        Sballvx = 10;

    else if (4.5 < ballvxTemp || ballvxTemp == 4.5)
        Sballvx = 11;

    }

    return Sballvx;
}

int getDiscreteVVelocity(float ballvy)
{
    int Sballvy = 6;

    float ballvyTemp;

    ballvyTemp = ballvy * 2;

    if (-1.5 <= ballvyTemp && ballvyTemp < -.5)
        Sballvy = 5;

    else if (-2.5 <= ballvyTemp && ballvyTemp < -1.5)
        Sballvy = 4;

    else if (-3.5 <= ballvyTemp && ballvyTemp < -2.5)
        Sballvy = 3;

    else if (-4.5 <= ballvyTemp && ballvyTemp < -3.5)
        Sballvy = 2;

    else if (ballvyTemp < -4.5)
        Sballvy = 1;

    return Sballvy;
}

String getAction(int xpos, int xvel, int yvel)
{
    string[] actions = new string[] { " 515", " 515", " 515", " 515", " 515", " 455", " 415", " 415", "
    " 415", " 515", " 515", " 455", " 415", " 415", " 415", " 515", " 515", " 455", " 415", " 415", " 415", "
    415", " 515", " 455", " 315", " 315", " 315", " 415", " 415", " 445", " 215", " 215", " 215", " 315", "
    415", " 415", " 215", " 115", " 115", " 115", " 115", " 345", " 115", " 115", " 115", " 115", " 115", "
    315", " 115", " 115", " 115", " 115", " 115", " 315", " 115", " 115", " 115", " 115", " 115", " 315", "
    115", " 115", " 115", " 115", " 115", " 555", " 515", " 515", " 515", " 515", " 515", " 455", " 415", "
    415", " 415", " 515", " 515", " 455", " 415", " 415", " 415", " 515", " 515", " 455", " 415", " 415", "
    415", " 415", " 515", " 455", " 315", " 315", " 315", " 415", " 415", " 445", " 215", " 215", " 215", "
    315", " 415", " 415", " 215", " 115", " 115", " 115", " 115", " 345", " 115", " 115", " 115", " 115", "
    115", " 315", " 115", " 115", " 115", " 115", " 115", " 315", " 115", " 115", " 115", " 115", " 115", "
    315", " 115", " 115", " 115", " 115", " 115", " 555", " 515", " 515", " 515", " 515", " 515", " 455", "

```

```

415", " 415", " 415", " 515", " 515", " 455", " 415", " 415", " 415", " 515", " 515", " 455", " 415", "
415", " 415", " 415", " 515", " 455", " 315", " 315", " 315", " 415", " 415", " 445", " 215", " 215", "
215", " 315", " 415", " 415", " 215", " 115", " 115", " 115", " 115", " 345", " 115", " 115", " 115", "
115", " 115", " 315", " 115", " 115", " 115", " 115", " 115", " 315", " 115", " 115", " 115", " 115", "
115", " 315", " 115", " 115", " 115", " 115", " 115", " 555", " 515", " 515", " 515", " 515", " 515", "
455", " 415", " 415", " 415", " 515", " 515", " 455", " 415", " 415", " 415", " 515", " 515", " 455", "
415", " 415", " 415", " 415", " 515", " 455", " 315", " 315", " 315", " 415", " 415", " 445", " 215", "
215", " 215", " 315", " 415", " 415", " 215", " 115", " 115", " 115", " 115", " 345", " 115", " 115", "
115", " 115", " 115", " 315", " 115", " 115", " 115", " 115", " 115", " 315", " 115", " 115", " 115", "
115", " 115", " 315", " 115", " 115", " 115", " 115", " 115", " 515", " 515", " 515", " 515", " 515", "
111", " 455", " 415", " 515", " 515", " 515", " 515", " 455", " 415", " 415", " 515", " 515", " 515", "
455", " 415", " 415", " 415", " 515", " 515", " 455", " 315", " 315", " 315", " 415", " 415", " 445", "
215", " 215", " 215", " 315", " 415", " 415", " 215", " 115", " 115", " 115", " 115", " 345", " 115", "
115", " 115", " 115", " 115", " 315", " 115", " 115", " 115", " 115", " 115", " 315", " 115", " 115", "
115", " 115", " 115", " 215", " 115", " 115", " 115", " 115", " 115", " 444", " 515", " 515", " 515", "
515", " 515", " 355", " 415", " 415", " 415", " 515", " 515", " 455", " 315", " 415", " 415", " 415", "
515", " 455", " 315", " 315", " 415", " 415", " 515", " 455", " 215", " 215", " 315", " 315", " 415", "
445", " 135", " 135", " 135", " 215", " 315", " 355", " 155", " 155", " 155", " 155", " 155", " 345", "
155", " 155", " 155", " 155", " 155", " 315", " 155", " 155", " 155", " 155", " 155", " 255", " 155", "
155", " 155", " 155", " 155", " 255", " 155", " 155", " 155", " 155", " 155" };

```

```

    int myindex = (xpos - 1) * 11 * 6 + (xvel - 1) * 6 + yvel - 1;

    return actions[myindex];
}

}

```