Project Post-Mortem
Brown University, CS196-2 Final Project

cswepson, dndadush, jperez1, nmehta1

1. Introduction

1.1 Goals:

In the most recent versions of popular fighting games like Soul Calibur, Tekken, and Dead or Alive you can select the types of weapons a player can use, the stage, and costumes. This type of customization and personalization provides users with the ability to change the look and feel of players and fights but only on a very high and imprecise level.

While these versions of recent fighting games give users a great deal of freedom, users want more control of the actual game play and want to be able to customize the moves and skills of a player instead of using the same pre-packaged animations over and over. This has inspired the creation of fighting games like Virtual Fighter with different modes where you can train your own players by sparring to endow them with new moves, and by judging their abilities in replay.

Our game makes use of real-time physical simulation with ragdoll physics which is a type of procedural animation that has been traditionally used to replace static death animations prepackaged in most video games. The Training/Fighting Mode is also unique and our AI provides the ability for players to play against the opponent on their own and/or receive reinforcement from the user and create better winning strategies while performing unique moves custom built for each player.

1.2 Initial Vision:

Our initial idea was to create a game whose ultimate goal was to be able to design a fighter with custom moves and train the fighter to beat other custom fighters using real-time ragdoll physics simulation and an AI using a reinforcement based learning algorithm. Our game can be broken into two modes with one main engine. Each of these modes is necessary for achieving the goal of training a fighter.

The Engine:

Using the Dojo library the engine renders graphics, simulates physics and calculates the game state for the AI. Since real 3D biped physics will be difficult to implement we will use ragdoll physics to simulate marionette style motion. In conjunction with the physics engine which Dojo already has, the engine simulates "strings" which pull the character into an upright position. In order to move these ragdoll characters we simply apply forces to pull the parts in the proper direction.

Move Builder: Provides the users the ability to create their fighter's own moves

The goal of the interface is to be as simple as possible. To reduce the amount of work the user has to do to create moves we've simplified the interaction from that of a traditional 3D modeling program. To select a limb there will be a drop down menu with a list of all the parts. Once you have selected a limb there are sliders corresponding to the X, Y, and Z location of the limb. The coordinates will be relative to the character so that the user can rotate the camera around the character while they are manipulating its position. This allows for much simpler manipulation of the body, but does not sacrifice the level of precision required for complex move creation.

After the user is finished organizing the character in their desired configuration they can add another configuration to add to the move or they can chose a target limb on the opponents body to attack. An attack is specified by selecting an attacking limb, the limb the player will attack with, and the limb to be attacked on the enemy player.

Moves can also be combined with each other to make combination moves. The user can select two moves and merge them either sequentially so that one happens directly after the other or combine them as a double move (i.e. punch and kick simultaneously).

To preview a move click the "Play Move" button. This will simulate what the move will look like during fighting mode. After the user is satisfied with the move they have created they can click "Finish Move" to finalize the move and add it to the list of moves the player can use when fighting. At this point the user must give the move a name and give it voice and key mappings for fighting mode.

Fighting Mode: User can apply moves to fight an opponent/fighter learns how to fight

The interface for the fighting mode will be even simpler. Since most of the fight will be a simulation the only important information is player's score and the time left in the round (the two players fighting is a given). Most of the fighter's learning will happen on its own. The user can press keys which are mapped to specific moves or give a vocal command to suggest move the fighter might try, but ultimately the decision is up to the fighter itself.

To assist with the reinforcement learning the user can also positively or negatively reinforce what the fighter is doing either through voice or through the keyboard.

Each fighter will have a score which will increase based on the force and location of a successfully landed attack. The fighter with the highest score at the end of the round wins.

1.3 Team:

There were four team members involved in this project: Zeke Swepson, Daniel Dadush, Janete Perez, Neil Mehta. All of the team members had worked on large projects in other software engineering courses for various classes.

1.4 Development:

Dojo was the 3D engine we used for our project. We used QtkRadiant to create maps. Most of the development was done in Windows using Visual Studio and C++. We also used Microsoft's Speech Recognition SDK. For the interface we also used wxWidgets.

2. Things that worked:

2. 1) Speech Recognition

We successfully implemented speech as our primary input modality using the Microsoft Speech SDK. This was challenging because it required learning how to use Windows COM components using C++. Generally, one tends to access COM components through interpreted/scripting languages such as C# or VB in which all the initialization / event handling setup are done automatically. Doing this in C++ requires you to learn how Windows event handling works, which is somewhat of an arcane art.

The user can both reinforce his fighter as well as indicate the next move for his fighter to perform using speech. When the user says "Good" or "Bad" his fighter is respectively positively and negatively reinforcement. Each move is given a name by the user when he creates it. We use these names to bind the user's speech to appropriate moves.

We found that one major benefit of using speech is that the user can really let his emotions run wild while still effectively controlling the fighter. The user can yell, "Kick him in the face!" and the Speech Recognizer will still successfully identify the word "Kick". This adds another dimension of satisfaction for the user, and makes the game play feel more dynamic. There is also an element of novelty in using speech in games which helps draw the user in.

Speech is also useful because it allows the user to create as many moves as he wants without having to think about keyboard bindings for each of them. This saved us the hassle of needing to create and save key mappings, and instead only put the burden on the user to find a good name for the moves he created.

We also enable the user to use speech to control other parts of the game functionality. For example, the user can say "New Match" at any time to bring up the match selection screen, as well as say "Help" to bring up the help menu.

2.2) Artificial Intelligence: Base Structure

We implemented the AI in Kung Foo Tamagotchi using a reinforcement learning based algorithm. We made it highly configurable in its ability to take different types of game state information on which to learn on. We have two simple configuration files which determine which game states the AI will pay attention to during the learning process.

We used a different approach for storing Q-Values than what is traditionally used: instead of using a Q-Table we implemented a Q-Tree. The Q-Tree is a decision tree where the decision variables correspond to different game state variables. In our current implementation we construct a Q-Tree by first examining the `Distance to the Opponent`, then branching on the `Opponent's Approach Speed`, and finally branching on the `Opponent's Attack Target' to arrive at a Q-State. The benefit of this approach is that we can store Q-States as we see them (saving memory), and have the possibility of dynamically aggregating similar states together.

The two important concepts that come out when examining the Q-Tree are the variable branching order and the state resolution. The branching order just tells you in what order you will look at the variables as you descend the tree. In our implementation, we only implemented a static branching order, which means that you always look at the same variables in the same order. The state resolution tells you how finely one discretizes a game state variable. In our implementation, the range of our state variables gets cut up into buckets where each bucket's size equals the state resolution.

The Q-Tree implementation proved more than sufficient for our needs, but we actually hardly scratched the surface of what is possible with this implementation. Had we had more time, we could have attempted some interesting techniques such as varying the branching order and dynamically varying the state resolution. These techniques would potentially have allowed us to examine more state variables without horrendously slowing down the learning speed.

2. 3) Artificial Intelligence: Learning

Having successfully implemented the base structure for our AI we had to decide how to actually make it learn. What ended working rather well was using automatic rewards from the environment. One obvious goal for our fighters to accomplish is to get close to each other before attacking. Instead of making the user painstakingly train the AI to get closer from every possible state combination, we automatically reinforced the fighters if they got closer to each other. We cut off this reinforcement once the player's are close enough, so as to not confuse the AI with useless reinforcement. To get the fighters to actually hit each other, we reinforced them every time they successfully landed a hit by an amount proportional to the force of the strike.

Another traditional problem we faced was the balance between exploration and capitalization when choosing the next action to perform. We chose to tackle this problem using an adapted version of simulated annealing. In our version, the learning rate serves as the temperature, where the learning rate for every state is proportional to the number of times that state was visited. To perform the annealing step, we first find the highest Q-Value for the state. Next we select an action for that state at random, and perform it with a probability proportional to the difference between the Q-Value associated to that action and the best Q-Value. Letting A be the Q-value of our randomly chosen action, and B being the value of the best action, the probability of performing the action given that we've randomly selected it is: exp((A-B)/(K*L*B)), where K is a constant and L is the learning rate for the state. Analogously to simulated annealing, we lower the lower the learning rate of a state every time we visit it by letting L = L * dL, where dL is a constant < 1.

This approach yielded good results, and guaranteed that we always perform a good amount of exploration of the action space. For our AI it turns out to be better for it to explore more versus capitalize more because the fighter's strategy needs to adapt itself to the fighting style of very different types of players. By always exploring, we partially avoid the problem of a fighter overspecializing himself to fight one particular opponent.

We found that with this AI design, the fighters were able to learn some basic yet effective strategies. For example, our OneArm character learns how to become a counter attacker by waiting for the opponent to move in to attack before unleashing his fist of fury on him. Our Vinny character, whose is more of a long distance fighter, learns to keep his distance and use his lunge attack repeatedly to do damage. We have also observed a case where the fighters become scared of fighting, which generally occurs after one fighter is repeatedly pummeled by the other, and proceeds to flee (go backwards) continuously.

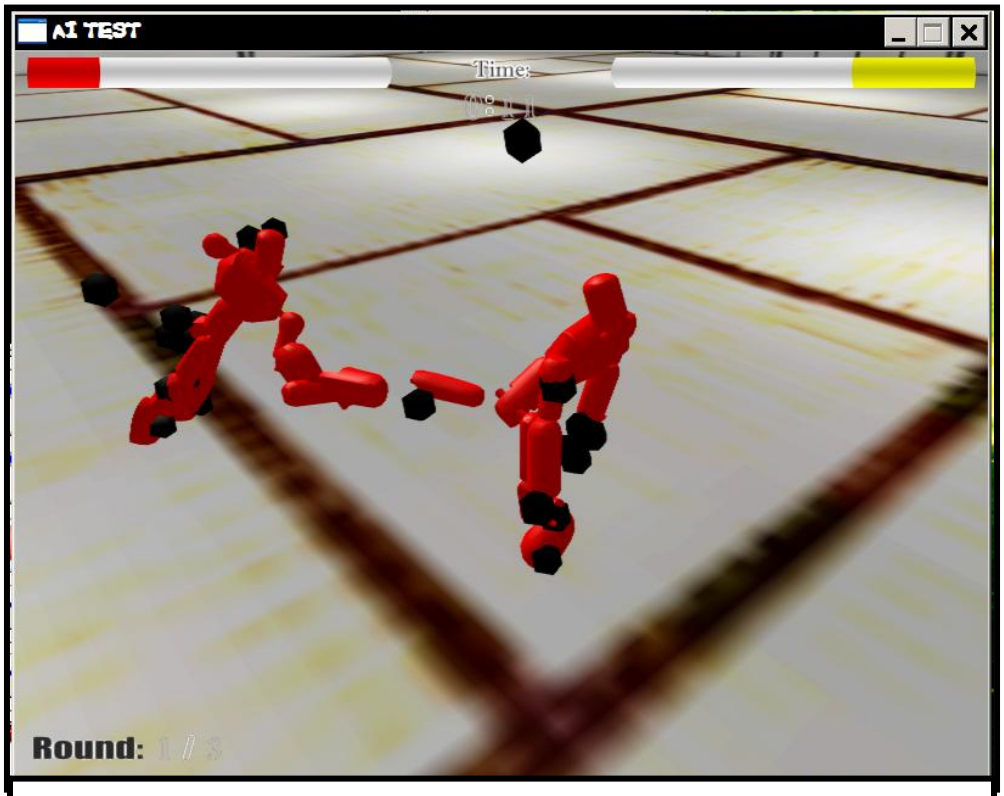

Figure 1:This is the fighting/training mode demonstrating two fighters with custom moves fighting each other and learning through our AI.

2.4) Marionette Physics

Setting up marionette physics was a probably the most painless part of the marionette physics engine. With very little effort we had a marionette that could remain standing upright on its own and even get up after being knocked over. After getting the stick figure to stand up, next we needed to get him to move like he was being pulled on by strings. This was a little more tedious and something we adjusted throughout the life of the project. For moving into different configurations a constant value for the force worked well, but getting attacking moves to achieve enough force that they were distinguishable from just one character bumping into the other was quite a struggle. In the end, because the marionettes don't hit each other very hard we amplify the force applied on collision. Because we know which character is attacking we check to see if this force reaches a certain threshold. If so it counts as an attack and we use the velocity to determine damage. If the attack doesn't exceed the threshold then it simply pushes the opponent away from it. This helps keep the marionettes from getting tangled up in each other.

We also went through several iterations of pull point interpolation. Initially, to move the marionette from pose to pose, we simply move the pull points to the desired position and the pull points would pull the limb with it. This brought up 2 issues. Transitioning from pose to pose was very jerky and didn't always achieve the desired results. Second, it was hard to tell when the marionette had actually gotten into position. The second problem was due to the fact that the limbs often overshot their destination because so much force was being applied to get them there and when they where there it was difficult to keep them in place. To fix this we decided to interpolate the pull point's position. Instead of jumping from one position to another we made them move over time. First we made them move some fraction of the difference between the pull point's location and the destination. To achieve the smoothest transitions from pose to pose we only calculate the step distance at the beginning of each transition.

2.5) Move Creation

Once the marionette physics was in a working state, move creation was easy. A move consists of a list of "configurations" or poses, an attacking limb and a target name. It knows which configuration it is currently trying to get into and whether or not it is ready to move to the next one. After each configuration has been done it sets the attacking limb's pull point to the attack strength and sets the position of the pull point to the position of the target. If there is no attack limb or target specified or the distance from the attack limb to the target is too great then it will not attack.

Configurations were simply a list with positions for each of the pull points. To maintain they would work no matter which way the marionette was facing we converted these positions into object space and convert them back to world space when they are needed.

In move editing mode it was very easy to write the code for selecting a pull point hand having it move first via the keyboard and then with sliders.

Moves sets are assigned by player id. Each player has a unique id and a name. The moves are stored in the players .move file. The file contains the id of the move, the name, the xyz coordinates for each of the pull points (in object space), the attack limb name (NONE if there isn't one) and target name (also can be NONE).

The hardest part of getting move creation working was getting the wxWidgets libraries working nicely with our code. After that it was a simple matter of learning how to use their system to achieve an intuitive user interface.

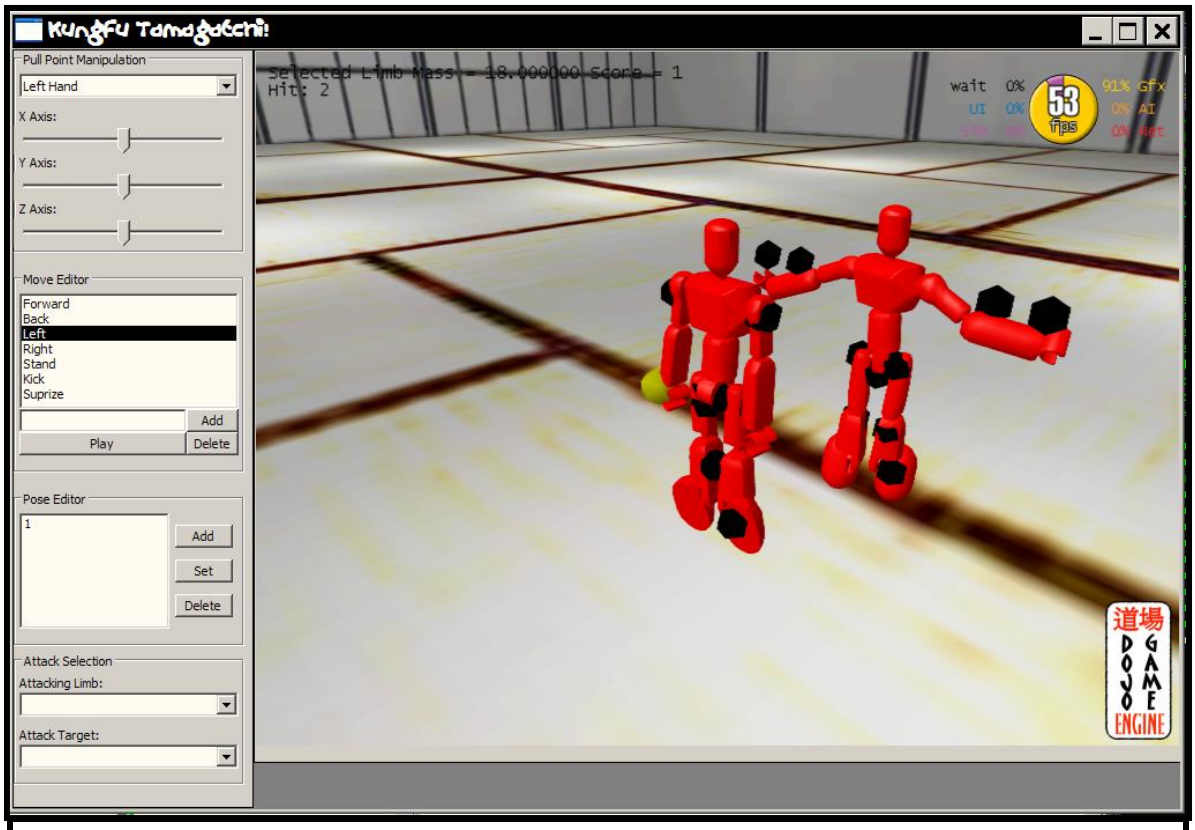

Figure 2: This is our move creator where custom moves are added to the fighters.

3. Things that didn't work:

3.1) Since we were using automatic rewards we got worried that the user's reinforcement would get unlearned too quickly. We can explain this by thinking of automatic rewards as continuous stimulus, and user reinforcement as a rare stimulus event. To fix this we attempted to remember how the user rewarded a particular action so that every time that action was performed it would again be rewarded by the same amount. This idea proved terrible in practice because it got the AI to fall into dumb cyclic behaviors. For example, if the user rewarded his fighter by 100 points to move forward from state x, and it only costs 10 points for the fighter to move back from state x+1 to state x, the fighter will have

the incentive to cycle continuously between state x and state x+1 since it gives him a 90 point guaranteed benefit. We therefore scrapped this idea, and let reinforcement continue as before.

What this made us realize was that each state had to essentially be accompanied by a valuation, and the automatic rewards for going from state x to state y should be valuation(y) – valuation(x). When the rewards are given in this way, we guarantee that no positive weight cycles appear inside the state space.

After changing our reinforcement strategy back to our original implementation, we found that appropriately reinforcing the fighter was very hard to do. It became clear that it was actually much easier to simply tell the AI what to do. Now this doesn't mean reinforcement is thrown out of the window, because the AI still learns the consequences of the actions it performs. Hence, if the user gives the AI a sequence of actions that result in a large payoff (e.g. hitting the opponent) the AI will be likely to remember that sequence in the future.

3.2) We were somewhat hoping that given the power of our AI architecture the AI would be able to learn some reasonably complex strategies. By complex, I mean things such as "Opponent is punching me from the front, so I should step to the right and then counter". This did end up being feasible for a couple of reasons.

First of all, though the use of Ragdoll physics simplified our lives in terms of move creation / move execution it made an accurate analysis of the user's physical state more difficult. The position of the torso, or the relative position of the arms, means much less when one is examining a Marionette than when one is examining a physically correct fighter. This means that there is no benefit in keeping detailed physical information about the opponent in the AI state, since it adds very little information while slowing down the learning speed tremendously.

Second, the unpredictability of the physics simulation made it very difficult for strategies to work consistently when the right situation arises. Hence, even if the AI pulls off exactly the right move and the right time, it may or may not connect dependent on the fickleness of physics. For example, there were bugs in Dojo which caused one or both players to freeze for an extended amount of time, thereby throwing off the observed benefit of a sequence of actions.

In short, the AI tended to find simple strategies that were good on average, instead of finding more fine tuned strategies based on more specific state information.

3.3) We attempted to create a system for causing the characters to fall if they were floating in the air, but ultimately it only half worked and didn't look very good. Initially we tried having the marionettes fall when they were hit with an attacking blow, this worked, but it looked awkward and unrealistic. Next, we tried multiple variations of detecting whether or not the marionette was touching the ground. Since most of the time the marionette only touches the ground occasionally we determine if its standing as an

average over time. This works, but two major problems arise. The first is that the character must now maintain some sort of balance. This gets tricky when the marionette is doing things like kicking. Often after a kick the marionette would fall over because it could not balance on one foot. The second problem which is some what related to balance is that the friction with the ground and the feet would cause the marionette to drag its feet behind it. This ultimately led to the marionette not having much use of its legs. In the end, we voted not to use gravity because of the aforementioned problems. It turned out to not be such a big deal. Because they are marionettes, it doesn't look bad and they tend to stay on the ground most of the time.

3.4) Initially we wanted the user to have very fine control over the motion of the character. What we found is that when working with a rag doll this is very hard. We didn't want to add too many constraints to the articulated model because we still wanted the motion to look more like a marionette than a real fighter. This would allow the user to create moves which would be more fun to see executed during fighting mode. Since the only control we maintained was over the position of the limbs it was very difficult to get the marionettes to bend limbs (especially arms) into specific positions. One thing we could have done, had we more time to focus on it, would have been to add some sort of intelligent torque control. Based on where we know we want a limb to be we could apply torques to the limbs to help get it into those locations. Probably through some sort of Inverse Kinematics system.

3.5) We had a great deal of trouble getting Dojo to behave itself in certain situations. One major problem we had was with time steps. We use the simulate time on simulation to control the speed of a move, but sometimes we would get bogus values which would often cause our game to crash. After much debugging and fixing other parts of the code we discovered the root of the problem was in fact not coming from our code. In the end we had to add several checks to ensure that if Dojo gave us a bad value we could recover. This reeked havoc on the AI because the AI would often get incorrect reinforcement for a significant amount of time.

Another major problem was that Dojo often continuously took up more and more memory as time progressed. This would often lead to our computers behaving very strangely and often crashing. We soon discovered that by simply minimizing any project related window immediately freed up a significant amount of memory. After finding that problem we began periodically minimizing the window to keep the program running. This worked most of the time and we saw that after time it would begin to manage the memory issue on its own. We're not sure why this happens, but one speculation is that Visual Studio steps in to cap the amount of memory usage. Unfortunately Dojo will still crash after a significant amount of time running (30 – 40 minutes). We were unable to determine the cause of this problem as well.

Quantitative Evaluation:

Move Creator:

User Interface:

The Move Creator is what allows the user to create new fighters with their own set of moves. The move creation interface utilizes our Pull Point system to allow you to control the position of the fighter's limbs. The user can select multiple Pull Points at a time to make different parts of the body move uniformly. The user can then use our slider system to control the movement of the Pull Points with fine precision. Once the user has achieved the desired position for all the limbs of his fighter, his can save this configuration as a pose. Each move is constructed from a sequence of poses that are smoothly interpolated through time. To create an attacking move, the user simply creates a standard move as described above but then adds an attacking limb and target at the end of it. The move is therefore interpolated as before, except that when the end of the move is reached the attacking limb is sent towards it target. Once the move is fully created, the user is able to see it in real time via the play button. The effect of attacking moves can also be observed in real time since there is a dummy Marionette within the interface on which to test them.

All in all, our Move Creation tool provides a highly functional system for complex move creation. Even though it may have a steep learning curve, we've found that once the system has been mastered moves can be created quite efficiently. Given all this, we believe that our Move Creation tool meets the specifications we set out to achieve.

Marionette Simulation Engine:

The Marionette Simulation Engine is what executes the moves for the fighter, and keeps him upright and facing the other player. It forms the base over which our game has been constructed. The Simulation engine supports moves with an arbitrary number of waypoints, allowing complex move creation. The engine supports attacking moves, which correspond to standard waypoint moves except with attacking limb / target pair appended at the end. This means the fighter can get his limbs in the proper position before performing the straight on attack.

After much testing, the simulation engine has been made as robust as possible (hacking around dojo bugs) and provides a base for a seamless game experience. With the complex move execution implemented, and robust performance we believe our simulation engine meets the specifications we set out to achieve.

Fighting Mode:

User Interface:

The User Interface for the Kung Fu fighting mode consists of a match selection interface and a heads up display for in round information. Our match selection interface allows you to set up matches between players created in our move editor with options such as the number of rounds to play as well as the length of each round. Our heads up display in fighting modes provides the following information: each player's health, the player names underneath the health bar, the moves available to player 1 (for speech control purposes), the current round, and the number of rounds won by each player. We also have a help screen, detailing the different commands available to you in game. All in all, we have a fully functional game interface for Kung Fu, which meets the specifications we set out to achieve.

Learning AI :

The Learning AI for our game has given us some very good results. Fighters can easily be seen to be learning over time and improving. The AI opponents are consistently able to be the human guided players, which validates the use of our automatic learning model. We have all spent a significant amount of time just watching the AI play against itself as it is capable of producing some pretty spectacular matches. From all of the above, we believe the AI has achieved its specifications for this project.