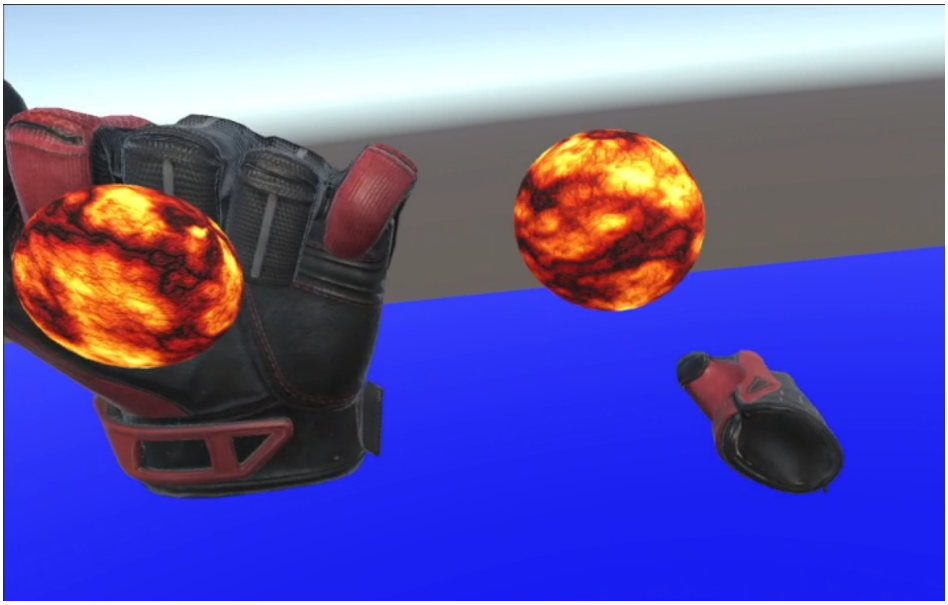
Classifying Virtual Reality Movements with Limited Examples

Matt Brown and Ben Kelly

May 3, 2021



Abstract

As virtual reality systems become more widely adopted and more developers look to develop for the new technology, a fundamental challenge of immersive game experiences is recognizing and classifying different player hand movements. Directly programming algorithms for recognizing certain motions is possible, but adding new motions is challenging and the approach does not scale well, especially for more complicated motions. Using a database of examples for a machine learning classifier is also possible, but gathering the necessary pool of data would be infeasible and expensive for students, small and even midrange developers.

We propose a lightweight, easy to use pipeline and classification algorithm that allows developers to record motions or use existing motion capture data as a set of examples that can than be used to detect different motions from live player data. This pipeline allows rapid development of motions by even single person development teams with reliable classification results for many types of motions.

1 Introduction

When it comes to the field of virtual reality (VR) new innovations are being made everyday. The problem that has plagued the industry is that it is too expensive in both man power and dollar bills to implement all of these innovations into one product. This is why the virtual reality market is dominated by big companies and there are few successful games coming out of its small indie game community. We are looking to even out the playing field in the VR development sphere. For our final project we created a pipeline that can be used to classify movements made by a player in a VR game. It is important to note that while this has been done before there are numerous flaws in existing systems. Two of these we are focusing on fixing in our pipeline:

- *No hard coding*: In most existing systems there is some type of hard coding going on. For example, to classify grabbing and moving objects many systems use collision detection. While this works for these motions it does not work for differentiating other motions. For instance these systems could not determine if the object was jabbed at or uppercut. On top of this these systems are not scalable for a large number of player movements.
- *Small Number of Examples*: For our pipeline we are looking to only use a small number of example movements to classify the live player movement. This is because producing a large number of example movements is not practical for many indie developers. While a complex machine learning algorithm might achieve a better result it would simply just require too many examples to be plausible.

With these 2 constraints in mind we created a pipeline that uses two separate comparison algorithms to classify movements. As you can see in Figure 1 our pipeline works by taking in a set of example movements and the live player data and returns the example movements that are most similar to the player's movement. In order to do this we first format the provided example movements to extract the necessary bone positions. Then during runtime we normalize the player's VR data to remove any noise that is created by the controllers. This normalization also makes the movement rotation independent so that it does not matter if the person was facing north or south. In the end a punch is a punch no matter which way you are facing. We can then input the example movements and the player's movement into both our distance algorithm and our own comparison algorithm. Both of these algorithms then output how likely it is that the player is making each example movement. Finally by combining these guesses we are able to classify a player's movements in real time with a pipeline that is feasible for a smaller game studios.

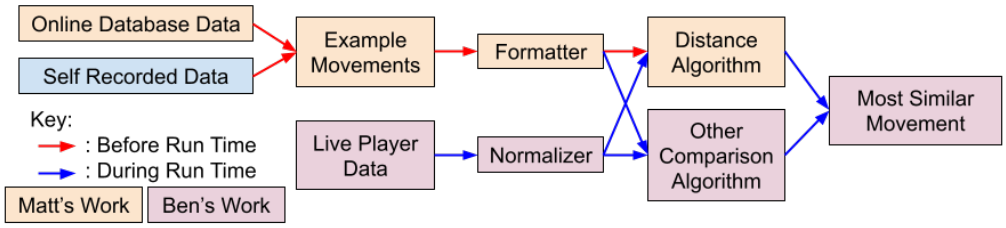


Figure 1: Our Pipeline

2 Existing Work

Surprisingly, while lots of research has been done on virtual reality in the past, there has been less work done related to hand movements and classification. Prior work in motion capture can provide insights in processing and working with the motion capture data we plan to use. Onuma et al. [6] is a particularly relevant example. They consider the creation of a distance function that allows the classification of movements for different movements in motion capture systems that do not depend on the number of frames in the motion. While this work is relevant the time independence is not something we can afford with a low data system like a simple VR setup. However, the general ideas behind classifying positions as one type of movement or another provides a basis for the work we will have to do with the motion capture data to isolate what we need from a set which has more data than is relevant as well as detecting outliers in the motion. Additionally, Lou and Chai [5] propose methods for denoising motion capture data for human movements, which will be an important task for transforming raw motion capture data into a more usable format for us. Keogh et al. [4] propose a new technique built on Dynamic Time Warping to normalize human motion capture data over global and local scaling as well as time scaling. This technique is used in their paper to index and edit motion capture data, but the idea of normalizing the data to fit many time scales and human sizes is particularly relevant for VR systems since the user could be a different size or move at a different speed than the original model, so simple comparing of locations would prove infeasible. However, while these works are useful to consider, the lack of data provided by VR systems means that considerations made for motion capture are excessive, and the algorithms are not applicable due since it relies on greater data and knowledge of each motion. For our work with motion capture data as example motions, we use publicly available datasets of human VR motions [1, 2, 3]. By using these databases we are able to get motion capture files and skeletal files that our users (the game developer) would produce. This saved us from the tedious process of making a perfect motion capture file for each of the motions we wished to test.

3 Virtual Reality Data Collection

The two ways we gathered data was through VR live data collection and existing motion capture databases. For the VR data, to record example movements we created a scene within Unity that would, when a controller button is pressed, record the global position of each hand controller until the recording is stopped with the same button. This data was then stored in a text file in order to be reused later. We can view this data with a scene that takes in a motion and plays back the data so that the data can be edited. This is useful for developers who might want to, for example, remove "dead time" where the controller is not moving at the very beginning and end of motion. Our goal with this method was to require the least amount of work and no preprocessing such that the data is easy to collect, easy to observe, and understandable by people.

The second form of VR data collection comes at runtime. This collection raises the fundamental question of how to collect the data for the live motion, since there is no obvious way to define the beginning or end points of a motion as data is being continously collected. This question was not the primary focus of our work, and we decided to collect a set amount of frames of data and used that fixed duration motion in our comparison. We remove this so called "dead-time" at the beginning and ends of motions by comparing the velocity of the hand at that point to a set ϵ. This method worked for our purposes, but limitations and potential for future work that could be done on this problem are discussed in section 9.

4 Example Movement Formatter

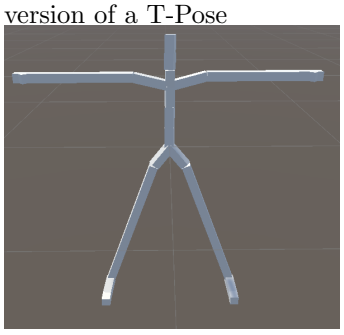
The first step of our pipeline is to collect example movements that the player's movement can be compared against. For our pipeline these example movements can be acquired in 2 different ways: they can be self recorded or they can be pulled from a motion capture database. In the case of a game studio they are able to use their self recorded motion capture/animation data that they are using for their NPCs' movements. Due to the scale of our project though we used movements from 3 different online motion capture databases [1, 2, 3]. Each of these movements came in the form of a .asf and the corresponding .amc file. While our formatter currently only accepts these file types it can easily be extended to accept other types too. Once the example movements have been collected they can be added to the pipeline which will start parsing and formatting them. It is important to note that we had to create our own formatter instead of using a prebuilt animator because developers are not able to retrieve the position of each bone when using an animator.

Figure 2: One bone in an .asf file

```
 id 2
 name lfemur
 direction 0.34202 -0.939693 0
 length 7.59371
 axis 0 0 20  XYZ
dof rx ry rz
limits (-160.0 20.0)
       (-70.0 70.0)
       (-60.0 70.0)
```

To talk about our formatter we first have to talk about .asf files and .amc files. Both of these files are used in tandem to represent one movement. A part of the .asf (aka the skeleton file) can be seen in Figure 2. This file represents the bones in a T-pose position as shown in Figure 3. We used 5 pieces of information in this file for each bone: its length, its T-pose direction, its axes, its degrees of freedom (DOF) and its children bones. While the length and the T-pose direction of the bone are self explanatory the axes, DOF and hierarchy are more complicated:

Figure 3: Our formatter's version of a T-Pose

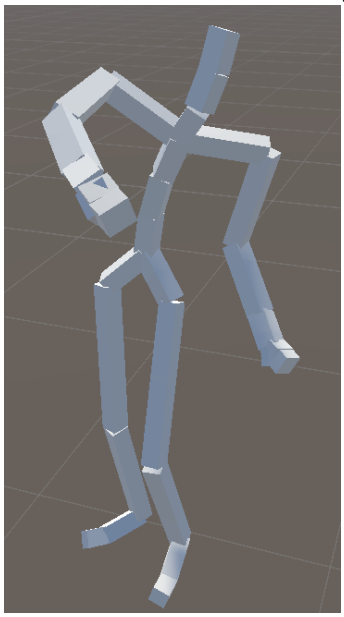


- *Axes*: The axes of a bone come in the form of a rotation of the standard x, y, and z axes in the x, y, and z directions. This new set of axes are then used to point the bone in its T-pose direction and to rotate the bone each frame.
- *DOF*: The DOF determines which ways the bones can be rotated. For example the knee joint can only be rotated along one axis while the shoulder joint can be rotated in along three axes.
- *Children Bones*: The children bones are affected by the movement of the given bone. For example the upper back bone is a child of the lower back bone because when you lean over and rotate your lower back the upper back and all of its children also have to be rotated and moved.

Figure 4: Part of a frame in an .amc file

```
1
root 9.62745 17.7973 -1.03923 -6.37909 -22.4014 3.49382
lowerback 5.60025 1.12798 -0.0431347
upperback 2.77139 1.49305 0.791281
thorax -0.311801 0.736113 0.749946
lowerneck -8.90502 6.57112 0.322245
upperneck 1.33699 8.87524 -3.9415
head 2.15388 4.30658 -0.892614
rclavicle -5.42682e-014 8.74653e-015
rhumerus -26.2308 10.8446 -85.4595
rradius 32.4455
rwrist -15.5297
rhand -23.7652 16.4224
rfingers 7.12502
rthumb 2.70415 -13.5402
```

Figure 5: Our formatter's version of a jab punch



With the use of just this file our formatter is able to create a T-pose that we can then animate using the .amc file (aka the motion file) part of which can be seen in Figure 4. In the .amc file there is the rotation of each bone along each of its DOF's for each frame. Our formatter uses these values to rotate the T-pose direction of the bone to determine the position and orientation of each bone at each frame which we can then use in our comparison algorithms. Our animated manikin can be seen in Figure 5.

5 Movement Data Normalization

A large portion of our work for the classification algorithm was developing a procedure for normalizing the VR hand data to be in a format which allows meaningful comparisons with example motions to be made. This procedure is necessary due to the shortcomings of using raw data of global hand positions. We describe the problems we encountered using raw data and the normalization step which corrected the problem.

- *Global position*: The data that is recorded is global hand data, but global position of the hands in a game is irrelevant. Instead, we transform the data to be positions relative to the starting position at the beginning of motion. The other option is to measure position relative to the player's body or head location, but this makes local position (i.e. where the hands are held relative to the body) important. This places limitations on the types of motion which could be classified, and is discussed more in section 9.

- *Player orientation*: Player orientation in the x and z rotational axes (with y pointing up) is generally irrelevant for VR motions. That is, the direction in which the player is facing should not matter when considering the motion the player is making. To fix this, we normalize the data by transforming it into a new basis. This basis is defined by three orthonormal vectors defined as follows, where *begin* is the first position in the motion and

end is the last position in the motion:

$$\hat{b_1} = \frac{\begin{pmatrix} end.x - begin.x, & 0, & end.z - begin.z \end{pmatrix}}{\begin{pmatrix} end.x - begin.x, & 0, & end.z - begin.z \end{pmatrix}}$$

$$\hat{b_2} = \begin{pmatrix} 0, & 1, & 0 \end{pmatrix}$$

$$\hat{b_3} = \hat{b_1} \times \hat{b_2}$$

By transforming the points into this basis, we remove dependence on the x and z axis rotations from the motion, but leave the y axis rotation so that motions which go up are not considered the same as motions that go down or do not change height at all. There are cases where height should not be considered, but for most motions height probably does matter, which is why we chose to retain that in this normalization process. This procedure did leave a problem where motions that moved hands toward the player body were treated the same as motions moving away, so punches could be performed by moving one's hands back into their body. This feature was not ideal, so we make an adjustment that if the location of the hand is closer to the body at the end of the motion than at the beginning, we multiply $\hat{b_1}$ by -1, and then calculate $\hat{b_3}$.

- *Differing distances*: Motions that travel different distances can be difficult to compare due to the noise that exists within VR systems. The data can have small changes in hand position, especially when moving slowly or holding the hand still. Therefore, to help remedy this we first find the coordinate with the highest absolute value within all the positions in the motion, then go through and divide each position by this maximum absolute value. This makes the motions go approximately the same length and also helps to scale down the effect of noise.

- *Differing durations*: Comparing motions over different duration is very difficult, so first we choose the duration of the longer motion, and then adjust the other motion so that the durations match. Given a target duration t' and an initial duration t_0, the i^{th} frame of the adjusted motion is the linear interpolation between two frames in the initial motion, $\lfloor \frac{i \cdot t_0}{t'} \rfloor$ and $\lfloor \frac{i \cdot t}{t'} \rfloor + 1$. The weight of the first, w, is the fractional part of $\frac{i \cdot t}{t'}$ and the weight of the second is $1 - w$.

- *Differing framerates*: Finally, motions captured from other sources such as motion capture databases might be recorded with a different framerate, which causes problems when trying to compare the frames of the different motions. Thus, we modify the motion with lower framerate to have the framerate of the other motion. This procedure is similar to the method described for changing duration. For the new motion, the i^{th} frame of the new motion happens at time $t = \frac{i}{f'}$, where f' is the new framerate of the motion. The new position is the linear interpolation between the frames in the initial motion which are closet to the same time, $\lfloor t \cdot f \rfloor$ and $\lfloor t \cdot f \rfloor + 1$ where f is the initial framerate. The weights of these frames are also the fractional part of $t \cdot f$ and $1-$ the fractional part of $t \cdot f$, respectively.

After this procedure, the two motions are normalized such that similar motions will be aligned and move together, allowing the comparison algorithm described in section 7 to operate successfully. Figure 6 shows the path of two jab punch motions before the normalization procedure is applied. Figure 9 show the motions after being normalized. Difficult to see is the time normalization, but in this example the red hand travels much slower than the blue hand. However, in the normalized motion, both end at the same time and in roughly the same location.

Figure 6: Unnormalized motion for an observed and example right-handed jab

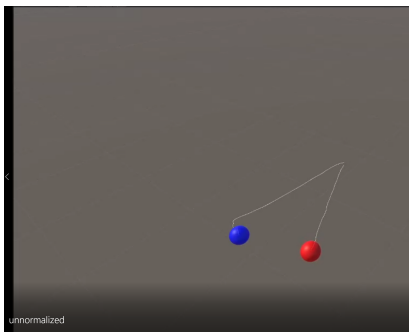
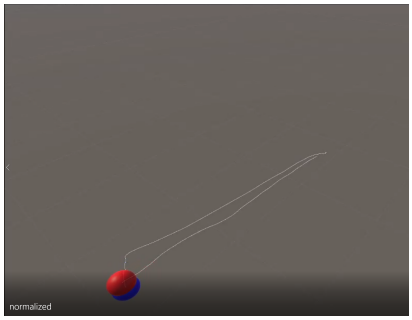


Figure 7: Normalized motion for an observed and example right-handed jab



6 Distance Algorithm

Once we have the normalized player movement and the formatted example movements we are able to start the classification process. Our first comparison algorithm is an extension of the distance algorithm proposed by Onuma et al.[6]. This algorithm calculates the average kinetic energy (KE) of each bone for each of the movements. This is done for each of the example movements as well as the player movement. The KE for example movements can be calculated before run time while the KE of the player's movement has to be done in real time. To determine a movement's KE our distance algorithm computes the velocity of each bone at every frame by finding its change in position. These velocities can then be averaged out to determine the bone's average velocity throughout the entire movement. This way the comparison at runtime is length of movement independent and therefore much faster which helps us keep a higher framerate. We can then store these average velocities in a vector until all the movement's have been processed. At run time, once the player's KE has been determined, our distance algorithm uses the Euclidean Distance Algorithm to determine how similar the player's movement is to each of the example movements. This gets us a good result but because the live player movement only has the positions of the hand bones we are only able to compare these 2 bones using this algorithm. On top of this the distance algorithm does not take into account direction. Therefore we need to use another comparison algorithm to sure up the weaknesses of this algorithm.

7 Comparison Algorithm

We describe the algorithm for classifying an observed motion. Let T be the set of all classifications possible. Let M be a table that maps a classification to a list of example motions. Each example motion is a list of 3-dimensional vectors representing positions. We describe the algorithm in pseudocode in Algorithm 1. For each of the possible labels, we go through all the example motions provided for that label. For each, we first use the distance algorithm to determine if the motion is a reasonable match based on distance. This allows us to then perform the normalization procedure, which removes total distance as a parameter among the other improvements on the raw data discussed in section 5. Once the data has been normalized, we can begin to compare frame by frame. Since both motion durations are the same and both have the same framerate due to the normalization, they both have the same number of frames. From there, we start with a high granularity, comparing the endpoints of the motions. Then, we half the step between keyframes until we do a comparison on the frame by frame basis. For each comparison we do, we look at the direction in between the keyframes we are considering for both the example and observed motion. If the dot product of these two is below a certain input threshold, we declare them as not matching and halt the comparison. At this point, we know to what level of granularity we can compare the motions such that they continue to match. With this information, we could many things, such as weighting the probability of each motion matching by the step, or choosing the motion with the smallest step. For our purposes explained in section 8, we simply use a cutoff that determines whether to accept a motion as the same. If the motion's step is below or equal to the cutoff, we add to a list of possible motions. While returning multiple answers might seem unhelpful, example motions that are similar enough to be a valid classification for the same observed movement are likely to be similar enough such that a player might find it challenging to mimic each motion independently without actually giving a motion which is ambiguous. This phenomenon is discussed in section 9 more, but it means that motions should be distinct enough from each other that an average player would be able to perform both clearly for the algorithm to work. However, that requirement is necessary for enjoyable gameplay anyways, so we do not think this weakens our algorithm's results on tractable example pools.

Algorithm 1: Classification Algorithm

input *observed*: the observed motion ;
input δ: the frame step cutoff ;
input c: the dot product cutoff ;
init *possible_motions*[ ] ;
forall $label \in T$ **do**
  forall $motion \in M(T)$ **do**
    if $DistanceAlgorithm(observed, motion) =$ ***false*** **then**
      break ;
    end
    init *observed_normal*[ ] ;
    init *example_normal*[ ] ;
    $NormalizeMotion(observed, motion,$ **ref** *observed_normal*, **ref** *example_normal*) ;
    $step \leftarrow |observed\_normal| - 1$;
    while $step > 0$ **do**
      $frame \leftarrow 0$;
      $success \leftarrow$ **true** ;
      while $frame + step < |observed\_normal|$ **do**
        $dir_o \leftarrow observed[frame + step] - observed[frame]$;
        $dir_o \leftarrow \frac{dir_o}{||dir_o||}$;
        $dir_e \leftarrow motion[frame + step] - motion[frame]$;
        $dir_e \leftarrow \frac{dir_e}{||dir_e||}$;
        if $dir_o \cdot dir_e < c$ **then**
          $success \leftarrow$ **false** ;
        end
      end
      if $success =$ ***false*** **then**
        break ;
      end
      $step \leftarrow step/2$;
    end
    $step \leftarrow step \cdot 2$;
    if $step \leq \delta$ **then**
      possible_motions[ ] $\leftarrow$ *label* ;
      break ;
    end
  end
end
return *possible_motions* ;

8 Results

We tested our system in three different ways. First, we recorded example motions in virtual reality and used the comparison algorithm to verify that recordings of the same motion would be detected and that recordings of different motions were considered different. We tested four different motions: a jab punch, an uppercut punch, a vertical circle, and a horizontal circle. Each was recorded with about 5 or 6 motions, and we then compared the different motions within the same motion and with each other. We used 0.9 for our dot product cutoff, and 2 for our frame step cutoff. For all of these motions, every motion was able to be recognized as the same as the other motions of the same type, and every motion was recognized as different that all other example motions of different classifications. This result was very exciting and we then moved onto testing the recorded VR movements with example data pulled from motion capture.

When only using the example movements and the distance algorithm we were able to determine the correct movement that was being performed by the player on a regular basis. As you can see in table 8 we rank each movement in terms of a likeness score which represents how probable it is that the player is performing that example movement. This likeness score is the distance between the two movements which is determined using the distance algorithm. By having several variants of each movement we are able to achieve better results. Of course if given more examples the classification will produce more accurate results, but for many game developers it is not practical to have more than on average about 5 or 6 variants of each movement. We are then able to use our likeness scores in combination with our own comparison algorithm to very accurately classify each of the player's movements.

Movements	**Left Hand Energy**	**Right Hand Energy**	**Likeness Score**	**Rank**
Player Movement	0.2616	0.6045		
T-Pose	0	0	0.6587	19
Jab	0.1559	0.4603	0.1788	3
Jab	0.1034	0.4482	0.2223	8
Jab	0.0974	0.5144	0.1872	4
Jab	0.1147	0.4814	0.1916	6
Jab	0.1082	0.4763	0.1998	7
Jab	0.0975	0.514	0.1873	5
Jab	0.1007	0.4485	0.224	9
Jab	0.1662	0.5855	0.0971	1
Jab	0.0985	0.5322	0.1783	2
Jump	0.211	0.1907	0.4169	11
Jump	0.2359	0.1123	0.4928	17
Running	0.8665	0.9708	0.7073	20
Walking	0.7318	0.1841	0.6309	18
Walking	0.2778	0.1524	0.4524	14
Swordplay	0.148	0.1983	0.4218	12
Swordplay	0.1567	0.1446	0.4717	15
Swordplay	0.153	0.1247	0.4919	16
Swordplay	0.1573	0.2211	0.3973	10
Swordplay	0.156	0.1881	0.4295	13

Figure 8: Distance Algorithm Results for a Jab

Finally, we developed and present a proof-of-concept for incorporating this data into a game scenario. A link to a video demonstration of this can be found at this link. For this demonstration, we recorded 5 distinct motions with the VR recording described in section 3. These motions were: left handed jab, right handed jab, left uppercut, right uppercut, and a "center" motion where hands are brought together. These motions were recorded 5-7 times each, the recording of all 5-7 examples only took about 1 minute. This is an extreme example of our algorithm since in real life a larger example set could easily be made if only a little bit more time was spent on gathering example data. Our goal was to show that even under this very low amount of time spent on gathering data, we can obtain reliable results in our demonstration. This would allow very quick development of motion based actions for VR systems and fast prototyping so that the developer could begin to playtest the player experience extremely quickly, a process very

important especially in VR games.

For the observed data, all of the motions we recorded were of length of approximately 1 second or a little bit less, so we decided to record the last 50 frames (1 second) of hand position data and use that in the classification algorithm. This approach worked for our relatively similarly timed motions, but does have limitations discussed in section 9. The dot product cutoff was set to 0.9 and the frame step delta was set to 2. As shown in the video, we were able to consistently recognize our 5 motions as distinct when they occurred and to recognize them as distinct from each other. One detail we noticed about collecting live data is that in a single motion performed by a user, there are many time steps during the motion when the last second of hand data could be classified as one of the motions. Thus, instead of doing an action whenever a motion we classified, we simply disabled an action for a "cooldown" period to avoid duplicate classification, which worked well for our purposes. Additionally, motions were labelled as "left-handed," "right-handed," or "both-handed." For a single-handed motion, only the motion data for that specific hand was used in the comparison. For the dual-handed motions, both hand data was used in the comparison. This allows us to have the left and right hands act independently and perform different single-handed actions at the same time, while also allowing actions which require both hands to be used to be recognized as well. We believe this demonstration shows our pipeline and algorithm's ability to work within a real-time game scenario while maintaining the ease-of-use and rapid development speed which we tried to capture in this project.

Figure 9: Example of a person using our demo to throw an uppercut and jab punch

9 Future Work and Limitations

There are several limitations and improvements that can be made to this algorithm. Some limitations are less of our algorithm and more with the problem and restriction of a limited sample size, and some could be overcome with changes to our data format, normalization procedure, and classification algorithm.

First, we discuss some practical limitations to the problem which it would be challenging to overcome with any algorithm. The intentionally limited sample size makes it so that motions that are very similar in general motion are challenging to recognize as different. The more similar the motions, the more examples would be required to build an example pool sufficiently big such that the motions would be well defined enough to differentiates between the two. In our algorithm, the dot product cutoff and the delta frame cutoff can be made more strict with larger example pools since the examples will cover more possible cases. However, this effect has a limit and the example

pool required to make these values incredibly strict and still cover all possible user motions that should be classified is likely so high that the premise of "limited example size" begins to break down.

There are cases, however, which are possible to handle that our algorithm is not capable of handling. The first and most obvious is that with the exception of movements towards vs. away the body, the algorithm does not consider local position of the hands as a parameter, only relative position from the location at the beginning of the motion. This prevents classification of motions which require specific hand positions relative to the body. For example, let's say a developer wanted a player to raise their hands above their head for an action to occur. Our algorithm can easily detect the raising motion, but cannot differentiate from a motion that raises the hands from the floor to the player's hips vs. a motion that raises the hands from the hips above the head. Part of our reasoning for not including this case is that we wanted no "tagging" of motions such that the developer has a set of options that must be set for each motions such as "local position important." Additionally, our algorithm relies on the use of a couple constants, the frame step delta and the dot product cutoff. Changing those values can affect the results of the algorithm significantly, so determining reasonable values is important. Developing an algorithm with less or no constants like this required would be ideal.

Finally, there are a couple issues with the way we gathered the data during a real time game situation. As mentioned in section 8, we obtained the last 50 frames (1 second) of hand data and used that in the comparison. This method is limiting and has significant room for improvement. The most obvious problem with this method is that it makes it prohibitively difficult to detect motions of significantly varied durations. For example, if we want to recognize a 3 second and 0.5 second motion, do we record the last 3 seconds, 0.5 seconds, or another duration? Any duration you record for will likely either be too short to capture the full 3 second motion or too long to capture the 0.5 second motion, so both cannot be recognized. One solution could be to set some maximum and minimum time, and then compare all possible intervals of minimum to maximum duration over the last set amount of frames. This solution raises the time complexity of a comparison from $O(n)$ to $O(n^3)$, which could cause problems with running this algorithm in real time for large enough durations.

10 Conclusion

We have presented our pipeline and classification system for hand motions in virtual reality that allows for reliable results for many types of motion and allows for fast and low cost development of a limited set of example motions. The system allows developers to quickly develop VR games or simulations and also allows feedback from users can be gathered and incorporated into the example set, allowing fast prototyping and iteration that aides development time and cost.

References

[1] Carnegie mellon university - cmu graphics lab - motion capture library. *Carnegie Mellon University - CMU Graphics Lab - motion capture library.*

[2] Mocap database hdm05. *Motion Database HDM05.*

[3] Sig center for computer graphicsmulti modal motion capture library. *University of Pennsylvania - SIG Center for Computer Graphics - Motion Capture Database.*

[4] Eamonn Keogh, Themistoklis Palpanas, Victor B Zordan, Dimitrios Gunopulos, and Marc Cardle. Indexing large human-motion databases. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 780–791, 2004.

[5] Hui Lou and Jinxiang Chai. Example-based human motion denoising. *IEEE Transactions on Visualization and Computer Graphics*, 16(5):870–879, 2010.

[6] Kensuke Onuma, Christos Faloutsos, and Jessica K Hodgins. Fmdistance: A fast and effective distance function for motion capture data. In *Eurographics (Short Papers)*, pages 83–86, 2008.