Immersive Virtual Reality Utilizing Hand Gesture Capture as a Replacement for Traditional Controls

James L. Gibson and Duncan Anthony Coulter(✉)

University of Johannesburg, Corner of University and Kingsway, Auckland Park, Johannesburg 2006, South Africa
dcoulter@uj.ac.za
http://www.uj.ac.za

Abstract. Current virtual reality systems are more immersive than other traditional forms of digital entertainment, but the handheld controllers they use detract from what would otherwise be a truly immersive experience. This paper outlines a project implemented to overcome this issue by creating a spell-casting game in which everything is controlled by the user's hand gestures. The implemented game made use of optical capture of the user's hands using a Leap Motion controller attached to the front of a virtual reality headset. The captured gesture data was passed through a neural network for precise gesture classification in real-time, and the correct sequence of classified gestures created a spell effect in the game. The user was able to cast 10 different spells utilizing various combinations of a set of 14 different gestures.

Keywords: Virtual reality · Artificial Neural Networks
Gesture classification · Human computer interface

1 Introduction

With the emergence of virtual reality systems over the last few years, a whole new world of possibilities has opened up for the entertainment industry. The move away from viewing games through traditional screens to experiencing them as immersive, 3D environments has created many new opportunities for how these games can be designed and how the player will interact with them.

The goal of this project is to create a game in which the player's only control of the system is through the use of gestures. This eliminates all traditional controls from the system and forces the player to practice timing and dexterity in order to perform well in the game. The system must be simple enough for new players to grasp, while complex enough to allow for a degree of progression and mastery as players become more adept at performing gestures in sequence. Once the game is functioning as intended and if time permits, the game can be adapted for play on a VR platform, further increasing the immersion of the player.

© IFIP International Federation for Information Processing 2018
Published by Springer Nature Switzerland AG 2018. All Rights Reserved
Z. Shi et al. (Eds.): IIP 2018, IFIP AICT 538, pp. 284–293, 2018.
https://doi.org/10.1007/978-3-030-00828-4_28

2 Input Hardware

Biometric input systems have been around for many years, with most being applied for various security and access control purposes. With regards to biometric systems that gather movement data in particular, two main methods of capture can be used. The first is accelerometers, which are devices that measure their own acceleration in the x, y and z planes. These devices provide very accurate data when it comes to movements, but can only relay predicted positional data based on the movements they've recorded. Cameras, on the other hand, provide very accurate positional data in real-time, which can in turn be used to calculate movement data as needed. This makes them ideal for capturing smaller movements requiring more accuracy and finesse, such as hand gestures (Fig. 1).

Fig. 1. Binary hand images after feature extraction.

The Leap Motion controller is a device that provides data tailored to gesture capture in hands, rather than the whole body. [6] also made use of a game environment to test gesture recognition, specifically the action of picking up and putting down blocks. The game environment contained many of these blocks, requiring the user to pick them up using grabbing or pinching gestures and place them in certain groups or structures by releasing the gesture. It was found that the best results were achieved when the pinching threshold distances were larger.

Leap Motion has also been applied to the field of sign language, as shown by [5]. 560 gestures were captured from a group of 4 people and normalized. Half of those gestures were then used to train and Artificial Neural Network, and the other half were used for testing. These gestures were all related to the letters of the alphabet in sign language, and the system scored an overall accuracy rating

of 96%, scoring perfectly on 21 of the letters, with the 5 outliers being gestures that looked too similar to one another.

Each of these methods of gesture capture and recognition have their merits and optimal situations for use, but as this project will focus on gestures performed with the hands rather than the whole body, the Leap Motion controller will be the best source of data capture. It has been proven to provide accurate, real-time readings of the coordinates of hand features and skeletal structure.

3 Model Overview

The core functionality of this project is based in biometric inputs from the user in the form of hand gestures and can thus be loosely modelled on the standard model of any biometric system. Furthermore, the interface that ties capture and classification representations together must accurately relay visual data back to the user in order for them to use the system effectively. The first section of the model is capture, where data will be acquired in real-time before that data can be used to advance user interactions with the system. Following this, the captured data must be refined so that elements other than the gesture being captured are excluded. This will eliminate all elements in the background. From this a set of features can be extracted that can be used to uniquely identify each gesture. These features must be normalized in order to properly identify the same gesture performed by different people with varying hand shapes and sizes (Fig. 2).

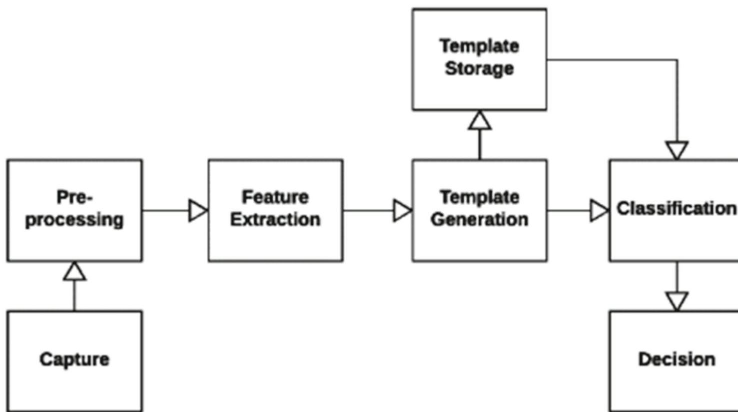

Fig. 2. High-level dataflow model.

Once an initial set of gestures has been captured and refined to a set of normalized features, it can be used to train a classifier for later use. This classifier will be used later to determine what gestures are being read in real-time when the user is interacting with the system. Different types of classifiers have

different outputs after training, and these outputs act as "experience" acquired from processing the training data. This experience must be stored for use when classifying later inputs, whereas the actual gesture templates used in training do not necessarily need to be stored.

The final section of the biometric model is the decision point. This accepts data received from the classifier and makes a decision based on that data. In this case, that decision will be whether or not the user has performed a specific gesture well enough for the system to accept it. The outputs of each decision will be relayed to the user visually through the interface. This interface must provide an accurate, real-time representation of the user's hands in order for them to ensure that their gestures are being represented correctly, as well as obvious responses from the system as triggered by the gestures they make. Each gesture must elicit a unique response form the system in order for the user to tell whether or not their capture has been correctly identified.

4 Implementation

Possibly the most important aspect of the entire system is how gesture data is captured, as any small inconsistencies in capture could cause major classification issues later on, bleeding into training and classification. The Leap Motion controller has proven to be a very reliable and accessible means of capturing hand data in real-time, and the libraries handling its use provide simple and easy access to the data it acquires.

The interface of the system takes the form of a game in which the player performs gestures to form spells as someone would form a sentence out of words. Certain gestures define the spell's core functionality while others act as unnecessary, but situationally useful modifiers. This allows players to alter spells to suit their needs at any point and take more time to prepare more powerful spells when they are not rushed. Before spells are discussed in depth, how the gestures that form them are captured must first be examined.

Initially, the tasks of training data capture and classification capture were performed by two separate interfaces. The Leap Motion libraries provide access to the X, Y, and Z coordinates of the center points of four bones per hand in relation to the position of the Leap motion controller, effectively covering both the capture and pre-processing functions of the biometric model. These positions were depicted as black squares during training data capture in order to provide the user with assurance that their hands were being captured correctly by the system.

Upon beginning the training capture process, the user would have a few seconds to move their hands into the correct position before any data was recorded. The system would then record the positional data of all 40 bone centers across both hands for 1000 iterations, usually over roughly 30 s. This data was then run through the feature extraction process, which calculates and stores the distances between each bone pair (Fig. 3).

Fig. 3. Leap motion representation of the hand [4]

Possibly the most important aspect of the entire system is how gesture data is captured, as any small inconsistencies in capture could cause major classification issues later on, bleeding into training and classification. The Leap Motion controller has proven to be a very reliable and accessible means of capturing hand data in real-time, and the libraries handling its use provide simple and easy access to the data it acquires.

This specific type of feature extraction was chosen because it focuses on the hands in relation to each other only, effectively disregarding the positional differences caused by any movement of the Leap motion controller itself. The feature extraction process redefines each gesture as 780 float values ($40 + 39 + 38 + \ldots + 2 + 1 = 780$) that do not experience any significant changes when a gesture experiences translation or rotation.

This feature data will still vary greatly based on the size and shape of the hands performing each gesture. To combat this, the data must be normalized. This normalization takes the form of a very simple equation:

$$z = \frac{x - m}{s} \tag{1}$$

In this equation, x represents a single feature value, m represents the mean of that particular feature across all of the training data, and s represents the standard deviation from that mean for that feature. This will shift the position of the mean to 0 and group the data more closely, making it easier to identify outliers. The mean and standard deviation data for each feature are stored for later use when normalizing newly captured data.

This normalized data can then be used to train the classifier, which in this case is a neural network. As far as neural networks far concerned, the architecture

of this one is fairly simple, beginning with an input layer of 780 nodes (one for each feature).

The middle layer made use of 11 nodes, each of which was connected to each of the input nodes, forming 8580 connections between the two. This layer is known as the “hidden layer”, as it is never interacted with directly by external sources. It is important to note that this hidden layer does not need to have the same number of nodes as the input layer, and can in fact be made up of multiple layers instead of just one. This hidden layer made use of the softmax activation function, calculating a float value between 0 and 1 for each node. The third and final layer is the output layer, which also consisted of 11 nodes, one node for each unique gesture, each of which were connected to each node in the middle layer (Fig. 4).

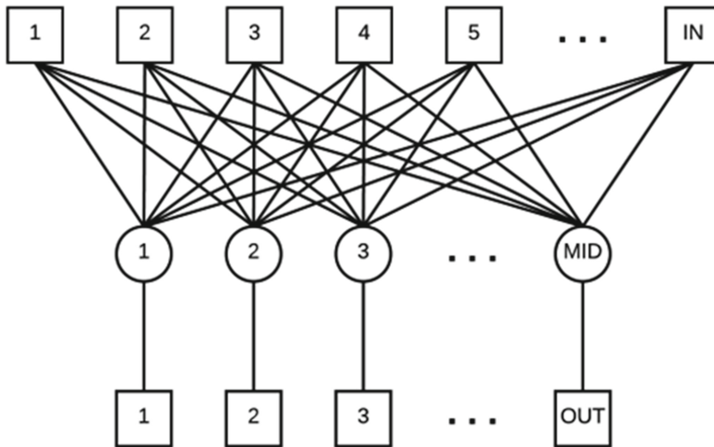

Fig. 4. Neural network architecture

During the training process, the neural network runs through 5 epochs. This means that all of the training data is fed through the network 5 times in order to refine it further, along with the expected value for that training data (i.e. an array of distances and an integer value denoting which gesture those distances should be recognized as). The actual process of refinement can be attributed to the alteration of weights and bias values associated with the connections between nodes. These values are the previously mentioned “experience” of the network, allowing it to perform classifications on data outside of the training set. These values are then serialized in JSON format alongside the architecture of the neural network for use in the system’s classifier, allowing them to be transferred between Python and C# implementations easily.

The classification interface was designed in Unity, displaying 3D models of the user’s hands, as well as the effects of the spells they’ve cast or are currently casting. Unity allows for entities modelled as GameObjects (a unity subclass) to

by easily added and removed from the "scene" that defines the game space. These GameObjects can be edited on an individual basis, defining properties such as physics interactions, colliders, and personal behaviour scripts. The models of the user's hands are made up of a number of GameObjects representing the palms and bones in each finger. These are grouped together and acted on in real-time based on the captures from the Leap Motion controller. It is important to note that these hand models mirror the user's hands based on the raw data acquired from the controller, rather than on data that has been through the process of feature extraction, normalization, or been passed through the classifier.

Due to compatibility issues between Unity and the packages needed to build the trained neural network from the JSON file, the classifier was built outside of Unity. The two run in parallel, streaming data between each other. At this stage, Unity captures gesture data and runs pre-processing, feature extraction and normalization on that data. It then writes the normalized data to a csv file and signals that new data is available. The classifier reads from the csv file and passes the data through the neural network, outputting an array of confidence values for each gesture between 0 and 1. These values are then written to a csv file and the classifier signals that confidences are available. Unity then reads in these confidences and finds the highest, which denotes which gesture was most likely captured. The sum of all confidences always equals 1, so one gesture will almost always stand out as the most likely option. Unity then adds thresholding to the most likely confidence, only accepting it if it has a confidence value of 0.9 or more (90% confidence). Using this architecture, the system effectively ignores the template generation and storage used by most biometric systems in favour of a trained neural network that contains weighted values assigned to classify any gestures passed through it.

When the system captures a gesture that passes classification and thresholding, it checks if that gesture is the first step in a larger "multigesture". These multigestures form a tree-like structure, with the first step being the root node and each new node being another gesture that must be matched along a particular path. If a capture matches the root node of any predefined multigesture, that multigesture is added to a list of currently active ones. If no root nodes are matched, the system checks if the capture matches any current nodes in any of the multigestures currently active. Once all gestures within a multigesture have been matched down to a leaf node, the spell element that the multigesture relates to is triggered, adding its effect to the currently active spell. This prevents spells form being cast accidentally, as the user will have to perform a very specific set of gestures before they trigger any action. Completing a multigesture also clears the list of active ones, ensuring that the user performs each correctly from start to finish.

These spells require a number of core elements before they can be cast, the first of which defines whether the spell is offensive or defensive. This must then be followed by adding an elemental effect to the spell. These elements work in a rock-paper-scissors format of countering each other, causing offensive fire spells to deal extra damage to defensive frost spells, for example. Spells can also have

any number of non-core modifiers added to them, increasing their damage, area of effect, duration, travel speed, and a number of other attributes (Fig. 5).

Fig. 5. Spell gesture tree

With the final gesture set reading and classifying accurately on any person's hands, the next step was to assign these gestures to spells and effects in order to create fluid casting sequences that could be performed easily and unambiguously by any user. The previously discussed gesture tree was used to track the user's progress towards a leaf node by performing each sequence in order. Once a leaf was reached, the tree was reset to the root and the ensuing spell was cast in the game.

5 Results

After several rounds of implementation and refinement, the process of 11 gestures, 1000 variations, 900 training rows, and 100 testing rows was repeated. The test accuracy averaged at 99.75%. The neural network was implemented in C# as a classifier and 9 of the gestures were recognized perfectly. Out of the remaining 2, 1 had an accuracy of slightly above 50% unless the hands had an almost perfect rotation, upon which it would move to high 90 s. The last gesture was not matched at all.

When refining the gesture set, it quickly became apparent that the more open-handed gestures were classified far more accurately than ones with more curled fingers. This was consistent among all of the people who supplied training data. The neural network differentiated similar gestures from one another well, leading to the inclusion of many of these slight variations in the final gesture set.

With regards to the neural network variations trained on the final gesture set, the variables that changed each time were as follows: Batch size, Number

of epochs, Number of hidden layers Out of these, changing the batch size made the biggest difference in terms of accuracy. These sizes ranged between 2 and 64, with 2 providing far superior results irrespective of how the other variables were adjusted. Changes to the number of epochs made very little difference once the number was over 2 or 3, with the training accuracy finishing in the high 90s after the first epoch and then never dropping below 99.98% for each epoch thereafter. The final number of epochs settled on was 5 and the final batch size was 2.

Adding more than one hidden layer to the neural network did almost nothing to improve accuracy as the accuracy was already incredibly high for the most part. Hidden layers with different activation functions, numbers of inputs and numbers of outputs all yielded the same result. Trained neural networks with more than one hidden layer reduced the speed at which the game could run, dropping the frames per second to an extent where the game was barely playable at some points. As such, only one single hidden layer was with 496 inputs and 14 outputs was used.

During training this neural network reached a maximum of 99.98% accuracy and performed incredibly well when tested on a wide variety of untrained users who had never seen the system before. Each of them could pick up and play the game with relative ease and casting spells worked fluidly and accurately (Fig. 6).

Fig. 6. In–game screenshot

References

1. Draskovic, D., et al.: A software agent for social networks using natural language processing techniques. In: Telecommunications Forum, Issue 24 (2016)
2. Jung, K., Kanga, H., Lee, C.W.: Recognition-based gesture spotting in video games. Pattern Recognit. Lett. **25**, 1701–1714 (2004)

3. Lavanya Varshini, M., Vidhyapathi, C.: Dynamic fingure gesture recognition using KINECT. s.l. IEEE (2016)
4. Leap Developer Documentation. https://developer.leapmotion.com/documentation
5. Naglot, D., Kulkarni, M.: Real time sign language recognition using the leap motion controller. s.l. IEEE (2017)
6. Pambudi, R., Ramadijanti, N., Basuki, A.: Psychomotor game learning using skeletal tracking method with leap motion technology. s.l. IEEE (2016)
7. Pinheiro, M., Kybic, J., Fua, P.: Geometric graph matching using monte carlo tree search. IEEE Trans. Pattern Anal. Mach. Intell. **PP**(99) (2016)
8. Xu, D., Xiao, X., Wang, X., Wang, J.: Human action recognition based on Kinect and PSO-SVM by representing 3D skeletons as points in lie group. s.l. IEEE (2016)
9. Yeh, W., Tseng, T., Hsieh, J., Tsai, C.: Sign language recognition system via Kinect: number and english alphabet. s.l. IEEE (2016)
10. Yin, Q., Wang, S., Miao, Y., Xin, D.: Chinese natural language processing based on semantic structure tree. s.l. IEEE (2015)
11. Zhong, H., et al.: A similarity graph matching approach for instance disambiguation. s.l. IEEE (2016)
12. Zhou, F., De la Torre, F.: Factorized graph matching. IEEE Trans. Pattern Anal. Mach. Intell. **38**(9), 1774–1789 (2015)