Using MS Kinect for real time gesture recognition

Real-Time System Project

Group 4: Pham Nhat Nam, Lam Phuoc Huy, Tran Anh Khoa
Instructor: Prof. Matthias Deegener

August 26, 2020

1 Introduction

Nowadays, human recognition has been an interesting subject for a wide variety of engineering projects because of its practical application in many industries. In 2010, a motion-sensing input device from Microsoft called Kinect was released and drew a lot of interest in this field due to its versatility and compactness. Thanks to its excellent capability, Kinect has been used to develop a wide range of applications such as gesture recognition, face recognition, body tracking for falling detection, voice recognition, etc. Moreover, due to its affordable price, it is now possible for both amateurs and students to participate in human recognition research and development.

In our team project, an computer application was made to receive input from Kinect device and output signal to VLC, a popular open source multimedia player, to control its basical functions. Because there are MS Kinects for Xbox 360 available in the storage, it is the main device for the project.

2 State of the arts

The study of gesture recognition have been carried out extremely extensively nowadays due to its practicality [1], ranging from hobbies, house-hold appliances such as gaming [2], theft prevention [3], to industrial projects like weather narration [4], healthcare [5]. Although the projects can be accomplished by many type of cameras which can collect real-time data (image, video), the development process is unnecessarily complex because the developers and researchers have to program various libaries to support for the data analysis and processing. This is the main reason for the rise of MS Kinect in this field due to the fact that every required functions and processes are integrated inside

and can be extracted throughout the Kinect software development kit (SDK). This issue will be discussed in more detail in the following section.

[6] gives an overview of the researches related to the Kinect, which proves the effectiveness of the device in the field. The popularity of the device into application can be seen in the variety number of books written in order to help beginners easily approach to the problem [7], [8], [9], [10], [11].

3 Kinect Introduction

As mentioned before, MS Kinect Xbox 360 is the main device for this project. Kinect is a line of motion sensing input devices produced by Microsoft and first released in 2010. It contains 3 cameras as shown in Figure 1. A normal camera for image capture and 2 IR components for depth sensing. It has widespread use in gesture recognition projects due to its ability to tracking up to 20 separate spots at the same time on the human's body and can build up the 3D environment with the help of depth sensing camera. It also includes a microphone array, along with software and artificial intelligence from Microsoft to allow the device to perform real-time speech recognition.

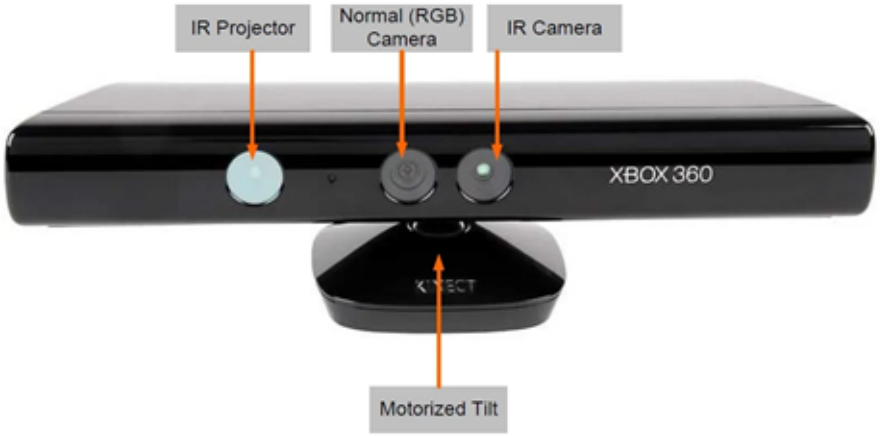

Figure 1: Components on Kinect Xbox 360 [12]

Kinect for Windows is a modified version of the Xbox 360 unit released on May 21, 2012, alongside the Software Development Kit (SDK) for commercial use. The hardware introduced better components to eliminate noise along with the USB and other cabling paths, and improvements in the depth-sensing camera system for detection of objects at close range. The SDK included Windows 7 compatible drivers for the Kinect device. Developers can use Kinect capabilities to build gesture recognition applications with C++, C#, or Visual Basic by using Microsoft Visual Studio 2010.

In November 2013 an upgraded iteration called Kinect version 2 was released. The biggest difference compares to the previous version is the higher resolution capability of the Kinect v2 camera. Even though the camera on the Kinect 1 device is a significant improvement over regular webcams at the time, the motion tracking ability is restricted by it's lower resolution output. The Kinect v2 included a 1080p camera and the ability to process 2 gigabits of data per second to read the environment. Therefore, the new Kinect has greater accuracy with three times fidelity of the previous version and even has the ability to see in the dark, thanks to a new IR sensor.

Figure 2: Kinect v2 for Xbox One [13]

It has a 60% wider field of vision and the data transfer rate is almost 10 times faster due to USB 3.0. For comparison, the Kinect v1 could only track 20 joints from 2 people while the Kinect v2 could detect 25 joints from 6 people. In addition, it is capable of detecting heart rates, facial expressions, and weights on limbs, along with much more extremely valuable biometric data. The table 1 below shows the detailed specification difference between two versions.

It is worth noting that despite its great improvements, Kinect 2 was considered a market failure compared to the Kinect for Xbox 360. One of the causes was the Xbox One required the Kinect 2 to be connected at all times to operate. This lead to a huge amount of criticism from the public regarding privacy and Microsoft decided to change this policy shortly after. Furthermore, no major game title released at the time could utilize Kinect capability in gameplay make it an impractical device for most gamers. As interest declined for the Kinect as the Xbox One went through its lifecycle, later hardware revisions of the console, those being the Xbox One S and Xbox One X did not include a port for the Kinect. Finally, Microsoft announced that the Kinect would be discontinued in 2017.

Features	Kinect v1	Kinect v2
Depth sensor type	Structured light	Time of Flight (ToF)
Red, Green & Blue (RGB) camera resolution	640 x 480, 30fps	1920 x 1080, 30fps
Infrared (IR) camera resolution	320 x 240, 30 fps	512 x 424, 30 fps
Field of view of RGB image	62°x 48.6°	84.1°x 53.8°
Field of view of depth image	57°x 43°	70°x 60°
Operative measuring range	0.8m-4m (Default) 0.4-3.5m (Near)	0.5-4.5m
Skeleton joints defined	20 joints	25 joints
Maximum skeletal tracking	2	6

Table 1: Comparison between Kinect v1 (Xbox 360) vs Kinect v2 (Xbox One)

4 Analysis and Implementation

4.1 Algorithm

Since the project is only available in short-term, complex algorithms such as Machine Learning, Deep Learning are out of consideration. Instead, applying mathematical calculations in a three-dimensional environment is done to classify the movement of body parts and make decisions. Each joint (tracking point) is designated by a 3D-coordinator set (x,y,z) in "skeleton space". The measuring unit in the set for the distance is defined in meters (if in depth value, milimeters is preferred instead) [10, p. 20].

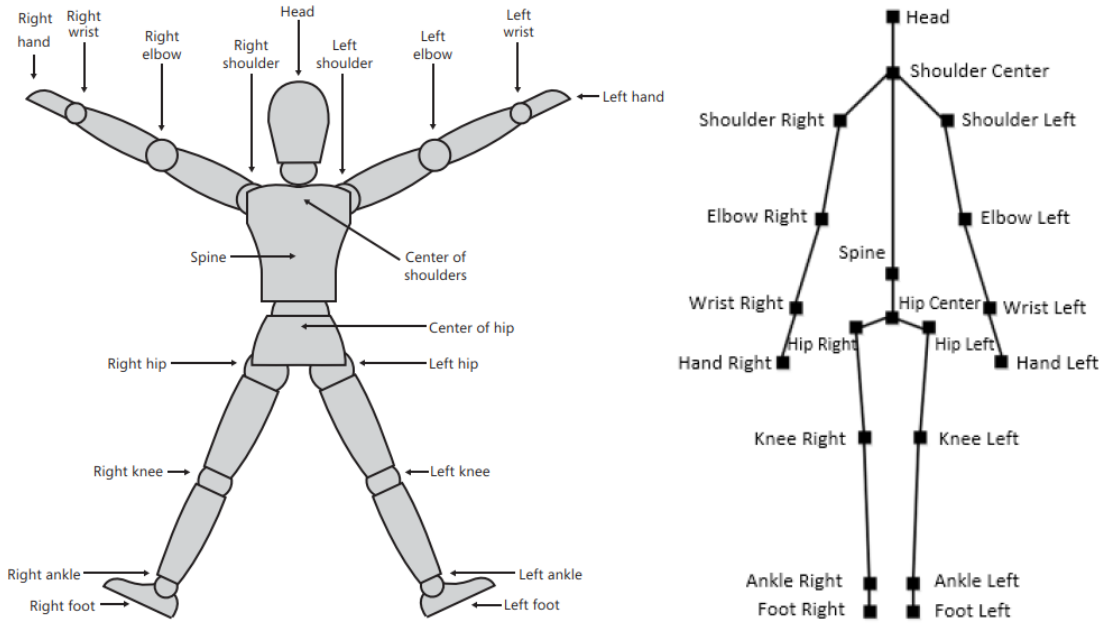

Figure 3: The 20 joints of a standing skeleton: left [10], right [9]

The whole 20 joints from head to the foot are monitored while user is in standing pose (Fig. 3) but in sitting pose, only 10 joints on two arms, the shoulder and the head (Fig. 4) are tracked.

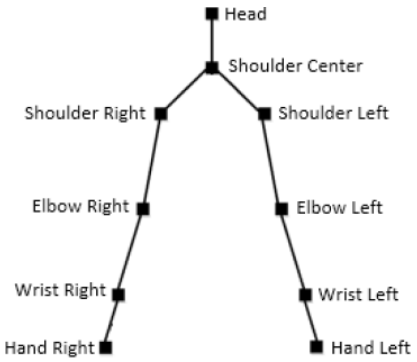

Figure 4: The 10 joints of a seating skeleton [9]

The kinect is in position (0,0,0) at default and the Z-axis heads toward the position of user in front of the device. In addition, the skeleton space follows right-handed coordinate rule which the positive value of X-axis increases to the right and the Y-axis positive values are in upward direction. Moreover, although the skeleton space is large which can cover a whole small room, the effective range is limited in order to have satisfactory tracking process. The suggestion for the limitation of the skeleton space is presented in [7, p. 113] as follow: X-axis varies from -2.2 to 2.2 meters; -1.6 to 1.6 meters is for Y-axis; Z-axis ranges from 0 to 4 meters (Fig. 5).

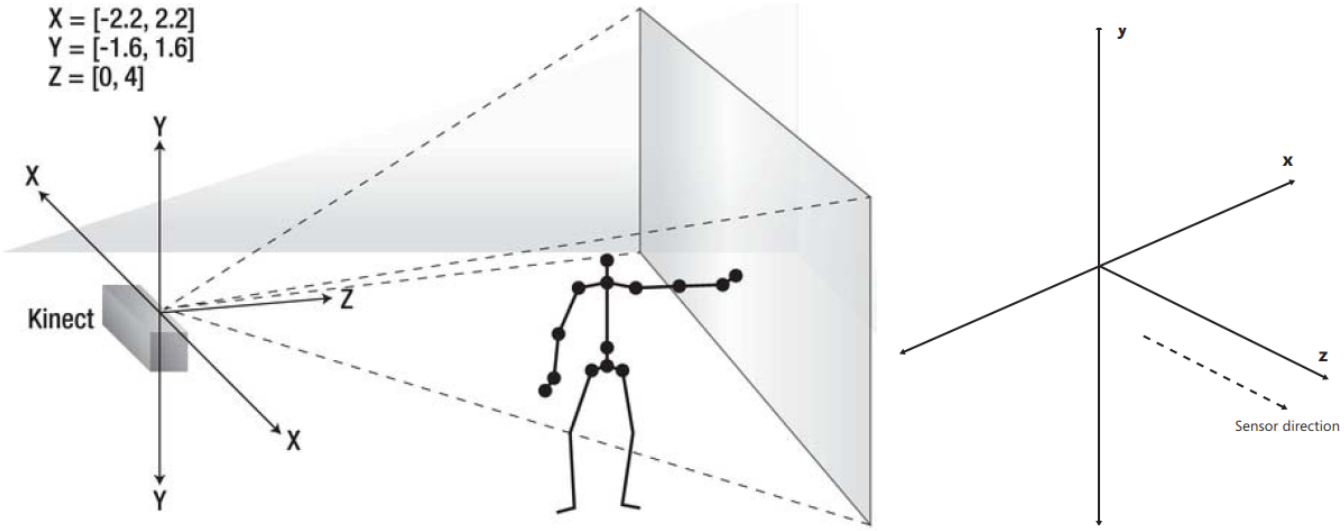

Figure 5: Skeleton space (left [7]), positive direction (right [10])

If applied to the algorithm to solve the recognition issue, the parameter change of the points in the coordinate system will be set according to an action. For example, when comparing a swiping-right action into the coordinate, it is clear that the x-axis coefficient

will increase and vice versa. Applying to the y-axis, raising hand action absolutely raises the y coefficient up while lowering hand reduces it. However, because the coordinate system in use is in 3D environment and the change of joints' coordinates are not always affected only one coefficient, it is unapplicable unless there is a solution to cover this weakness. From this point, a small trick is considered as a temporary solution: "gesture area", which is a subset space of the skeletion space where a gesture and its output is hard-coded into it. With this approach, there are more than one "swipe right", "raise hand" action which is not a good solution in solving gestures recognition problem but in application, it helps expand the collection of output to control the program as intended. To make it easier to understand, (Fig. 6) show an example of swiping action from left to right which provides an output of spacebar input into computer. The red sub-space in the image is the "area" which is identified by a set of coordinations defined by the coordinations of the skeleton joints. With the skeleton as the center of reference, the user's height will not affect the setup.

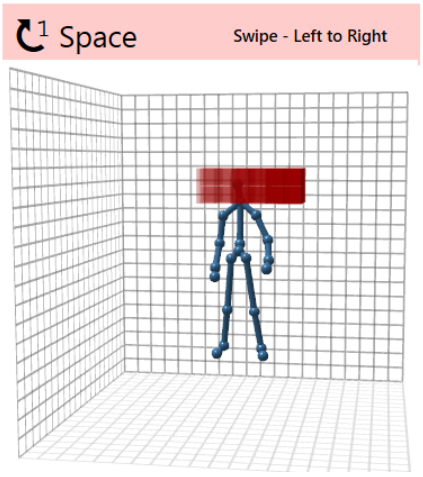

Figure 6: Example of "gesture area"

4.2 Implementation

4.2.1 MVVM Model

The solution chosen for this project is a software structural design pattern called Model-View-View-Model (MVVM). The model can be implemented effectively with Windows Presentation Foundation (WPF) Application and its .NET framework provided in C#. As illustrated in Fig. 7, MVVM consists of three main components, which are View, Model, and ViewModel [15]:

- View component represents the UI and visualization part of the project.
- Model component is where the algorithm and application variables being processed.

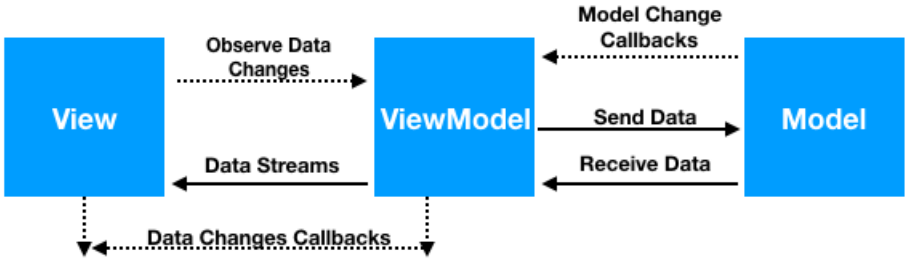

Figure 7: Example of MVVM [14]

- ViewModel controls the processed data and properties of Model to update their representation in View through a binding mechanic. In C#, the Extensible Application Markup Language (XAML) is responsible for this binding.

Some specific interfaces in C# required to implement this pattern, e.g. *INotifyPropertyChanged, ICommand, IValueConverter, IMultiValueConverter*. The *INotifyPropertyChanged* [16] interface signals any changes in data properties and updates them to the UI. In the program, it is the base class which represents a gesture and its properties. The *ICommand* [16] interface allows the handle of any buttons or clicks included in the UI designing phase. In addition, it provides an option whether the controlled buttons are clickable or not. The *IValueConverter*, and *IMultiValueConverter* [16] provide a conversion between data types of one or more variables. The design of UI is discussed in the next section.

4.2.2 GUI and Gesture Design

When starting the program, an UI in Manager Mode appears as illustrated in Fig. 8. Two options are available at this stage, which are "Add New Gesture" and "Load Gesture Collection". If a collection is loaded, "Play" option becomes available.
In **Manager Mode**, if one or more gesture is created recently, a "save gesture collection" option becomes available for the users, and the program writes a .json file to store the gesture properties for later use. If the users wish not to save the changes, the collection can discard by clicking "New Gesture Collection". In addition, if a specific gesture in the collection needs to be edited or deleted, "Edit Gesture" and "Delete Gesture" are available with two buttons next to the gesture name.

If "Edit Gesture" is chosen, the user enters **Editor Mode** as in Fig. 9. In this mode, several requirements have to be completed before saving the gesture to the collection. First, a name can be typed in for the gesture. Second, the joint type must be chosen to perform the gesture. There are 24 joints which can be chosen [16]. However, for the sake of simplicity, only 7 joints are available for the program, which are "Head", "Right Elbow", "Right Hand", "Right Knee", "Right Foot", and their left-side counterparts.
After choosing the joints, users also needs to specify the area where the joints hit to perform the action by clicking on the frame cells in the "Front View" as well as in the

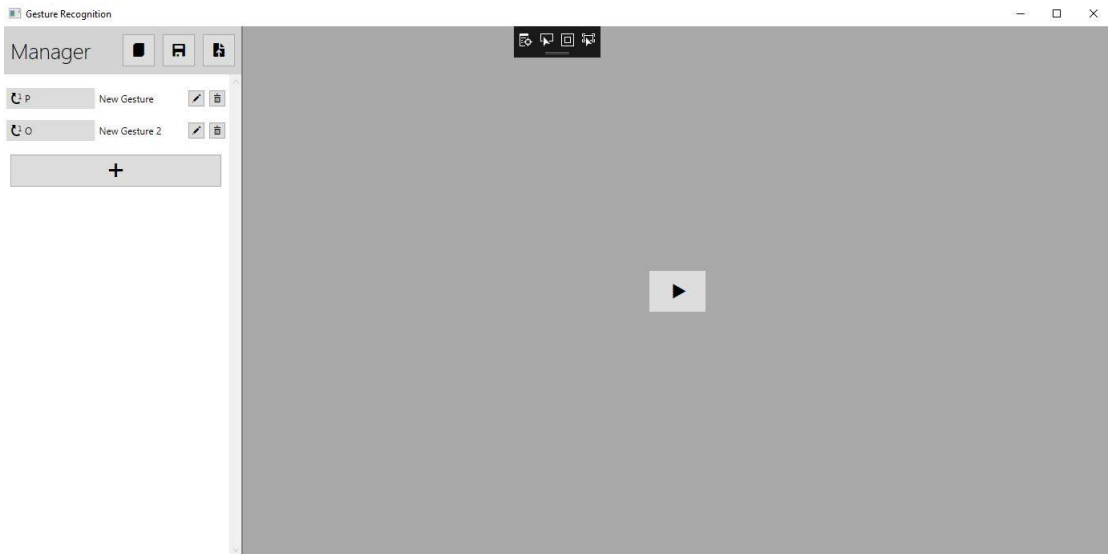

Figure 8: UI in Manager Mode

"Side View". Those cells are later called hotspot cells. Furthermore, to design a multi-frame gesture, users can create more frames with arbitrary number of hotspot cells. It is possible to switch the position of a frame with frames next to it by clicking on the "Move Frame Backward" or "Move Frame Forward" indicated with arrow icons. In Fig. 9, a 3-frame gesture is being edited. The users should also note that the choosing of hotspot

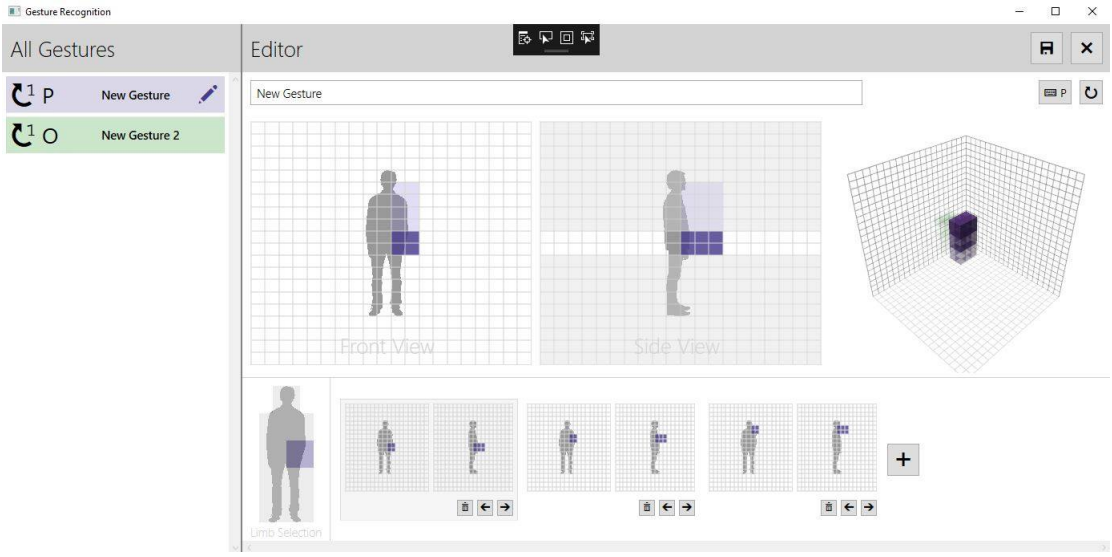

Figure 9: UI in Editor Mode

cells must be conducted in the "Front View" in order to logically create a boundary in the "Side View". After the choosing process is completed, cubic objects indicates where the joint should hit are created in the 3D grid on the right for simulation. Continuous gestures, i.e., gestures that create connecting hotspot cells, are preferred, since a user

only has 500ms to hit one frame and move to another. A normal human pose picture is added in the background in order to help the users to have an overarching view of the gesture being designed.
Last but not least, the users must specify the keyboard outputs produced by the gesture when recognized. The outputs can be a single or combination of buttons. Moreover, the users can choose if those buttons are pressed or hold by selecting the "Press/Hold" button.
Finally, after completing all the requirements, the "Save Gesture" becomes available should the users want to keep the changes or "Discard Changes" otherwise. By clicking either of the buttons, the users return to Manager Mode.

4.2.3 Algorithm explanation

As stated in section 4.2.2, the 3D grid is created by assemble 20*20*20 cubic cells, each with an edge of 15 cm. For the sake of simplicity, the center of the 3D grid is chosen to be the origin of the new system coordinates, i.e. the pivot point of coordinate measurements. However, the received coordinates of the tracked joints are relative to the Kinect sensor. Therefore, a simple calibration of the human body must be made before executing the algorithm. A user needs to position himself so that his center of mass matches the center of the 3D grid, and the tracked joints coordinates must be recalculated in the new coordinate system.
The idea of the gesture recognition algorithm is straightforward. For each gesture in the gesture collection, two variables which indicate the previous state and current state of matching are initialized to be "not matched". In case a hotspot cell in a frame is to be hit in real-time, the current state of matching is set to "matched". The program moves on to consider the next frame if it exists for the current gesture. The previous state is now set to "matched". If there is any matching regarding the next frame, the current state is update again. As the algorithm moves from frame to frame the value of previous state and current state are continuously updated until the last frame of the gesture is reached. In that case, if the tracked joint hit the hotspot cells, the current state is set to "confirm".
For the recognition confirmation, if the gestures contain only one frame, the values of "not matched" and "confirm" in the previous state and the current state, respectively, are enough to accept the gesture. In case of gestures with more than one frame, it is more complex for the recognition. The first condition is that the values of the previous state are "not matched" or "matched", and the value of the current state is "confirm". The second condition is that the total amount of time a user need to move to the next frame, after leaving the current frame, is no longer than 500ms. To be clear, a user can stay in the current frame as long as he wants to (with his predefined joint to match the hotspot cells), but after leaving it, there is only 500ms for him to hit the hotspot cell of the next frame in order for the program to accept that gesture as valid. In the program, an *enum* type variable with possible output values {In, Out, Confirm} are used to represent the states "matched", "not matched", and "confirm", respectively.

Algorithm 1 summarizes the proposed gesture recognition discussed above

---

Algorithm 1 Gesture Recognition

Input: Tracked joints received from Kinect sensor, a saved gesture collection file or a recently created one which contains appropriate gestures with a specific joint to perform the action.
Output: Recognize the human actions corresponding to the predefined gestures in the collection

1. function: Gesture recognition
2. **for** each gesture in the Gesture Collection **do**
3. **enum** jointStates {Out, In, Confirm}
4. Using the HipCenter as the origin of coordinates, calculate the relative coordinates of the tracked joint (in meters)
5. jX = (trackedJoint.X - O.X)*100
6. jY = (trackedJoint.X - O.X)*100
7. jZ = (O.Z - trackedJoint.Z)*100
8. Gesturei.PreviousState <= Gesturei.CurrentState
9. Gesturei.CurrentState = Out
10. **for** each frame in Gesturei **do**
11. **for** each cell which is a hotspot in the 3D grid **do**
12. **if** tracked joint matches a cell **then**
13. Gesturei.CurrentState = In
14. **if** the frame being considered is the last in Gesturei **then**
15. Gesturei.CurrentState = Confirm
16. **end if**
17. **end if**
18. **end for**
19. **end for**
20. **if** Gesturei has only 1 frame **then**

21. **if** Gesturei.PreviousState = Out & Gesturei.CurrentState = Confirm
22. **then** recognize the gesture performed
23. **end if**
24. **end if**
25. **if** Gesturei has more than frame **then**
26. **if** Gesturei.PreviousState = Confirm & Gesturei.CurrentState = Confirm **then**
27. **if** the latest tracked joint match occurs in the last frame of the gesture and not timeout
28. **then** recognize the gesture performed
29. **end if**
30. **end if**
31. **end if**
32. **end for**
33. end function

4.2.4 Visualization

Not only the keyboard outputs simulation being performed, the gesture recognition algorithm also incorporates visual effects such as highlighting the gesture recognized in the collection, and lightening the 3D cells with which the tracked joints hit in order to support the programmer during the testing phase as well as the designing phase. Those functions are summarized in the two tables below:

Possible Cases	Previous State	Current State	Action for Gesture
1	Not matched	Confirm	Highlighting the cells in current frame
2	Confirm	Not Matched	De-highlighting the highlighted cells since matching is no longer occured

Table 2: 1-frame gesture 3D appropriate cells visualization

For 1-frame gesture, as in table 2, case 2 agrees with the recognition algorithm on circumstance in which the gesture is accepted, and highlights all the appropriate hotspot cells. The corresponding gesture is also highlighted on the collection.

Possible Cases	PreviousState	CurrentState	Action for Gesture
1	Not matched	Not matched	No action needed
2	Not matched	Matched	If the frame is the 1st frame of any multi-frame gesture, highlight the cells. Otherwise, de-highlight any highlighted cells
3	Not matched	Confirm	De-highlight any highlighted cells
4	Matched	Not matched	
5	Matched	Matched	If the frame is the 1st frame of any multi-frame gesture, highlight the cells of the current gesture.Otherwise if the action is timeout (>500ms) or if the action order is not preserved, de-highlight highlighted cells.
6	Matched	Confirm	If the action is timeout (>500ms) or if the action order is not preserved, de-highlight highlighted cells. Otherwise highlight the multi-frame cells.
7	Confirm	X	De-highlight any highlighted cells

Table 3: Multi-frame gesture 3D appropriate cells visualization

It is obviously more cases in multi-frame gesture as shown in Table 3. Case 1, Case 4 and Case 7 are trivial.
Case 2 signals the start of a new gesture. Therefore, in this situation, the corresponding cells of the first frame are highlighted. The "if" condition is to force the gesture to be performed in the correct order, starting from the first frame.
In case 3, in which the highlighted cells are de-highlighted, the conditions are similar to those where a gesture being accepted. This is no contradiction, but a specification the authors find suitable. Only the hotspot cells of continuous gesture are highlighted. Fig. 10 demonstrates an example of case 3, in which a "green item" in the gesture collection is highlighted, with the hotspot cell in their normal 3D visualization state.

Case 5 points out the transition period of a gesture, e.g., matches recognition is being done in intermediate frames of a gesture. The condition in the table is to ensure that the action being performed occurs in the correct order and with time constraint (500ms is the maximum amount of time for each frame transition). For example, for a 4-frame gesture, moving backward from in frame 3 to frame 2 which results in "matched" both value of the previous state and the current state is not valid. Case 6 is the last stage of gestures which already reach Case 5. Fig. 11 shows a valid gesture with both the hotspot cells and the collection item being highlighted.

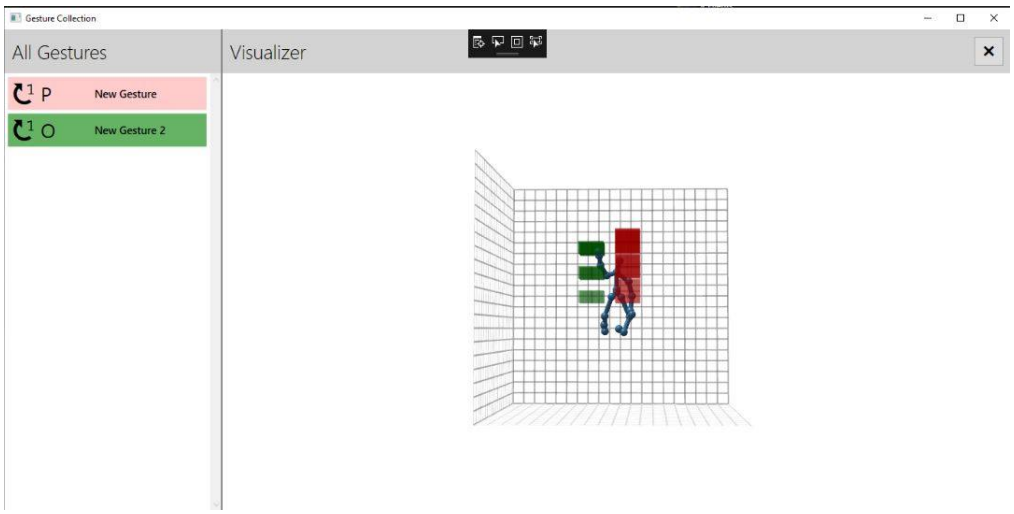

Figure 10: Gesture in the collection recognized without the hotspot cells being highlighted

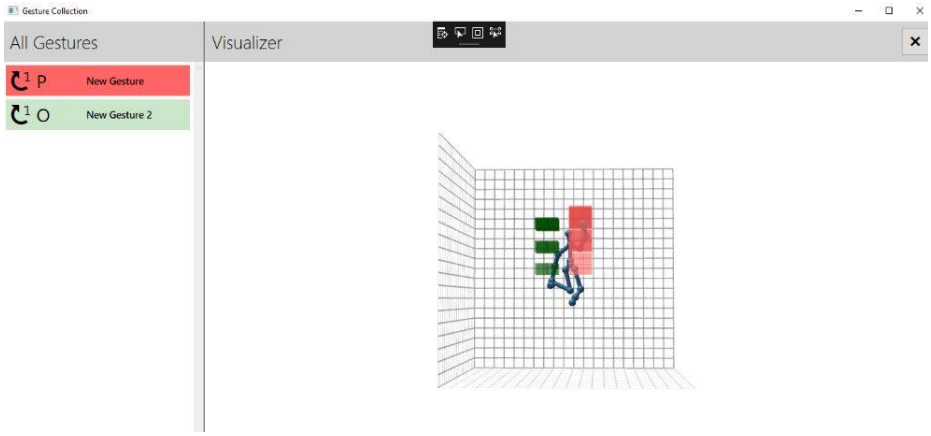

Figure 11: Gesture in the collection recognized with the hotspot cells being highlighted

5 Conclusion

This report shows that applying Real-Time System concepts can help us simplify the application development process and increase the reliability of our program. Even though there are still things that need to be improved like the ability to use a single gesture to execute dynamic operations such as turn the volume up or down, we can say that our final product satisfies all the uses cases initially planned. Through this project, our team members learn a lot of different algorithms and procedures that are necessary for Real-Time System development.

References

[1] T. Ma and M. Guo, "Research on kinect-based gesture recognition," in *2019 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*, pp. 1–5, 2019.

[2] C.-H. Tsai and J.-C. Yen, "The development and evaluation of a kinect sensor assisted learning system on the spatial visualization skills," *Procedia - Social and Behavioral Sciences*, vol. 103, pp. 991 – 998, 2013. 13th International Educational Technology Conference.

[3] R. Bavya and R. Mohanamurali, "Next generation auto theft prevention and tracking system for land vehicles," in *International Conference on Information Communication and Embedded Systems (ICICES2014)*, pp. 1–5, 2014.

[4] R. Sharma, J. Cai, S. Chakravarthy, I. Poddar, and Y. Sethi, "Exploiting speech/gesture co-occurrence for improving continuous gesture recognition in weather narration," in *Proceedings Fourth IEEE International Conference on Automatic Face and Gesture Recognition (Cat. No. PR00580)*, pp. 422–427, 2000.

[5] M. Khan, S. I. Ahamed, M. Rahman, and J. Yang, "Gesthaar: An accelerometer-based gesture recognition method and its application in nui driven pervasive healthcare," in *2012 IEEE International Conference on Emerging Signal Processing Applications*, pp. 163–166, 2012.

[6] J. Rodrigues, J. Rodrigues, P. Cardoso, J. Monteiro, and M. Figueiredo, *Handbook of Research on Human-Computer Interfaces, Developments, and Applications*. USA: IGI Global, 1st ed., 2016.

[7] J. Webb and J. Ashley, *Beginning Kinect Programming with the Microsoft Kinect SDK*. USA: Apress, 1st ed., 2012.

[8] S. Kean, J. Hall, and P. Perry, *Meet the Kinect: An Introduction to Programming Natural User Interfaces*. Apress, 1st ed., 2011.

[9] A. Jana, *Kinect for Windows SDK Programming Guide*. Community experience distilled, Packt Pub., 2012.

[10] D. Catuhe, *Programming with the Kinect for Windows Software Development Kit*. Developer reference, Microsoft Press, 2012.

[11] J. Howse and J. Minichino, *Learning OpenCV 4 Computer Vision with Python 3: Get to grips with tools, techniques, and algorithms for computer vision and machine learning, 3rd Edition*. Packt Publishing, 2020.

[12] Nick Davis, "Teardown tuesday: Microsoft xbox 360 kinect," 2017. Available at `https://www.allaboutcircuits.com/news/`

teardown-tuesday-microsofts-xbox-360-kinect [Last accessed August 24, 2020].

[13] Tom Warren, "You can now use kinect to log into windows 10 with your face," 2015. Available at https://www.theverge.com/2015/12/10/9885000/microsoft-kinect-windows-10-windows-hello [Last accessed August 24, 2020].

[14] Anupam Chugh, "Android mvvm design pattern," 2018. Available at https://www.journaldev.com/20292/android-mvvm-design-pattern [Last accessed August 26, 2020].

[15] J. Greene, J. Strawn, and R. Team, *Design Patterns by Tutorials (Third Edition): Learning Design Patterns in Swift*. Razeware LLC, 2019.

[16] Microsoft, *Kinect for Windows SDK*. Available at https://docs.microsoft.com/en-us/previous-versions/windows/kinect-1.8. Last accessed: August 5, 2020.